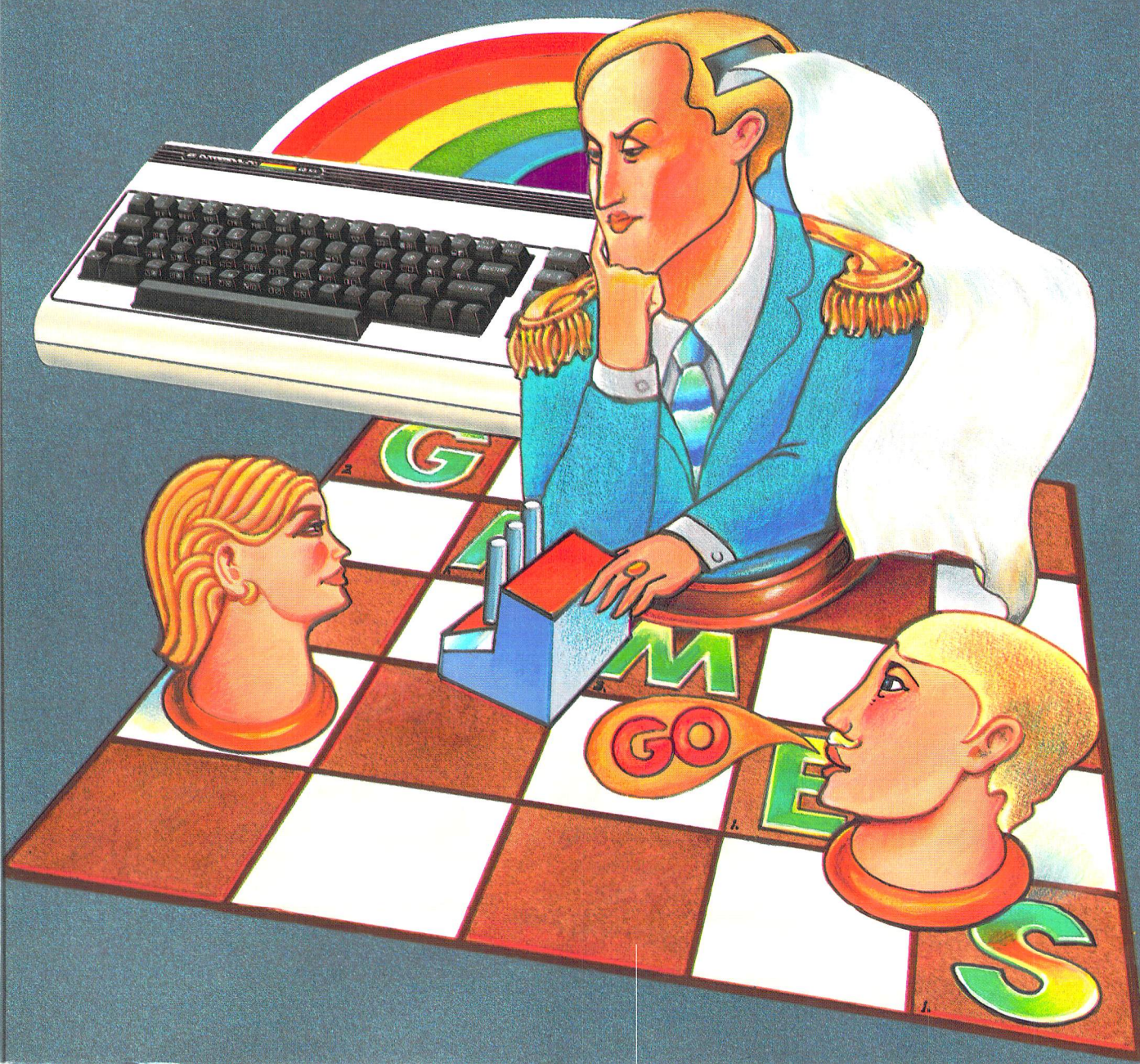


1923

COMMODORE 64™ ADVANCED GAME DESIGN

GEORGE A. AND NANCY E. SCHWENK



COMMODORE 64TM

ADVANCED GAME DESIGN

GEORGE A. AND NANCY E. SCHWENK



TAB BOOKS Inc.

Blue Ridge Summit, PA 17214

After Pearl is a copyright of SUPERware, Inc.
Defender is a copyright of Williams Electronics Inc.
PAC-MAN is a copyright of Bally Midway Mfg. Co.
DONKEY KONG is a copyright of Nintend.
Panzer War and Baseball's Best are copyrights of Windcrest Software Inc.

FIRST EDITION

FIRST PRINTING

Copyright © 1985 by TAB BOOKS Inc.

Printed in the United States of America
Reproduction or publication of the content in any manner, without express
permission of the publisher, is prohibited. No liability is assumed with respect to
the use of the information herein.

Library of Congress Cataloging in Publication Data

Schwenk, George A.
Commodore 64 advanced game design.

Includes index.

1. Commodore 64 (Computer)—Programming. 2. Computer
games. I. Schwenk, Nancy E. II. Title.
QA76.8.C64S36 1985 001.64'2 85-2802

ISBN 0-8306-0923-7

ISBN 0-8306-1923-2 (pbk.)

Cover illustration by Al Cozzi

Contents

Acknowledgments	v
Preface	vi
Introduction	vii
1 Overview	1
2 Playing Prime Time	4
3 Playing Word Square	8
4 Playing Takeover	15
5 Ingredients for a Good Game	22
6 Step-By-Step Game-Design Process	24
7 Designing a Simulation Model	31

8	Artificial Intelligence: The Computer Plays	41
9	Programming Considerations	59
10	Implementing a Game-Design: Professional Example Games	69
11	Prime Time Program Documentation	71
12	Word Square Program Documentation	86
13	Takeover Program Documentation	109
14	Making Money: Advice to Profiteers	128
	Appendix A Software Producers	131
	Appendix B Listing Conventions	132
	Index	133

Acknowledgments

We wish to thank Martha S. Coffey for her editing skills and her eye for detail; Judy Ng for her speed and accuracy at the keyboard; and our playtesters,

Eugene McNally, Kathy McNally, Mary Sweeney McNally, and Roy Woods, for their many helpful comments.

Preface

Both of us have been associated with mainframe computers professionally for many years. In our spare time, one of our favorite activities has always been playing board games. The microcomputer has changed our lives tremendously because we now possess a tool that we can use to design and play games of our own choosing. We have found that

designing games fulfills our creative needs and is more enjoyable than simply playing games. Now we spend considerably more time designing computer games than playing them. This book was written to teach you how to use your Commodore 64 to enter the exciting and rewarding world of computer-game design.

Introduction

If you enjoy games, then *Commodore 64 Advanced Game Design* is for you. This book is a tutorial on the microcomputer game-design process, and it also brings professional-quality games to the book buyer.

WHY SHOULD YOU BUY THIS BOOK?

There is something for everyone in *Commodore 64 Advanced Game Design*. Whether you are already hooked on computer games, or you have never played them and want to see how they differ from board-type games, you will enjoy playing the three games included. You will also save a considerable amount of money by purchasing this book, which adds three professional games to your game collection.

If you are curious about how computer games are designed, then this book will educate you by providing an in-depth look at this fascinating process. If you are interested in designing your own computer game, you will experience the thrill of seeing your own design come alive on your Commodore 64. We take you through the entire game-design pro-

cess step by step, using numerous examples from the three games included. If you complete an original game-design project and want to reap the financial rewards that a successfully published game can provide, then you can take advantage of the valuable advice, based on our own experiences, on profitably submitting your game for publication.

WHY IS THIS BOOK UNUSUAL?

Some books provide only game listings and others provide only programming instruction. *Commodore 64 Advanced Game Design* includes games that are sophisticated and professional—in sharp contrast to the simplistic games that are standard fare in the current book market. In addition, this book combines programming instruction and game-design instruction, and uses numerous professional games as examples.

WHICH SECTIONS OF THIS BOOK SHOULD YOU READ?

This book has been organized so that you can

use it in the way that makes the most sense for you. If you simply wish to play the three games, the instructional Chapters 2, 3, and 4 are all you need to read. On the other hand, if you are more ambitious and wish to learn the game-design process, you will benefit from tutorial Chapters 5 through 9 as well. These design tutorials and the program documentation, Chapters 10 through 13, should be particularly informative when you have passed the stage of learning simple BASIC coding and would like to design and write your own game programs. Newcomers to computer-game design should start at the beginning of the book and read all the way to the end. If you have previously tried to design your own microcomputer game and find yourself stuck, you can skim through the book until you find the specific application problem that you have been trying to solve.

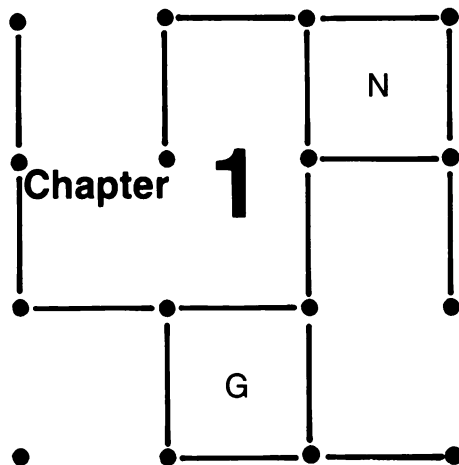
WHAT IS INCLUDED IN THIS BOOK?

Commodore 64 Advanced Game Design is available in two versions. One version includes a loader disk enclosed inside the back cover. The other version, which does not include the loader disk, comes with an order form for purchasing the loader

disk if you do not want to type in the program listings.

The three games included in this book appeal to people who enjoy nonviolent family fun in which the winner is determined primarily by skill rather than luck. Detailed playing instructions are provided, along with documentation and program listings. The chapters on the design process serve as a tutorial that goes from the conceptualization stage to the finished-product stage. This book may also help you make some money by suggesting how to turn your game into a professional product. In addition, this book contains two helpful appendices. Appendix A provides the names and addresses of publishers who accept computer games for publication from free-lance authors. Appendix B includes the conventions used to produce the program listings in this book.

If you plan to play the games and not merely use the game listings for illustrative purposes, you would be well advised to purchase the loader disk. If you wish, however, you can type in the listings as provided in Chapters 11, 12, and 13. Before beginning to type in these programs, be sure to consult the appendix on listing conventions.



Overview

You are a microcomputer owner. You feel comfortable with your machine, you have used it to play games, and you are familiar with BASIC programming. Now you are ready for a new challenge—designing your own Commodore 64 computer games!

WHY PLAY COMPUTER GAMES?

Many people buy home computers primarily for playing games. After we amassed a collection of nearly 100 board games, we bought our first home computer in 1981 and proceeded to purchase and write numerous games for it. We now have four home computers and nearly 200 computer games in our game collection.

We all know that computer games can have many advantages over traditional games. Consider a typical board game. Counting out paper money bills is tedious, for example. Performing mathematical computations in our heads or on paper is time-consuming and prone to error. Keeping time on your wristwatch or on a room clock is

haphazard. Keeping all players totally honest is also a difficult task. Undoubtedly the biggest problem of all is not always having someone to play with! When you bring out your favorite board game, does your wife suddenly have to wash her hair? Do the kids suddenly remember they have homework to do? The fact is that we all do not enjoy the same types of games.

Now consider the possibilities of computer games. There is no paper money to count. All mathematical computations are taken care of quickly and accurately by the computer. Timekeeping is performed internally, and there is a bell to signal that time is up. There is no way for someone to cheat—the computer sees to that! And the biggest problem of all is solved; you always have a playmate. Your computer will play with you at 2:00 A.M. when everyone else is asleep. It will play with you on a beautiful sunny day when everyone else is outside. It will even play with you when you are sick and everyone else is afraid of catching whatever it is you have. You may not yet be able to pack up the computer and take it to the beach, maybe someday . . .

WHY DESIGN YOUR OWN COMPUTER GAME?

Undoubtedly you have seen commercial computer games with serious faults, and you have said to yourself, "I could do better than that!" Maybe the computer plays too slowly, or perhaps the graphics are primitive. Using *Commodore 64 Advanced Game Design* as a guide, you will be able to create computer games that are better than many of the games currently on the market.

More importantly, perhaps, is the fact that you will be able to create a computer game about a topic that interests you—be it race car driving, fishing, or the defense of the Alamo. You will have the power to make your game as simple or as complex as you like, and you can set all the rules according to your own liking.

GAME DESCRIPTIONS

The three computer games included in this book are diverse and have broad player appeal. All three are educational, nonviolent, and can be played by one or more players. They are presented from a playing point of view from least complex to most complex. The first game, Prime Time, will be of interest to anyone who watches nighttime television. In Prime Time you assume the role of a network television executive making programming decisions, and your competition comes from the other two major networks.

In Word Square your goal is to form words from randomly generated letters while you race against the clock. If you think you have a good vocabulary, wait until you meet the "Wiz," the genius inside your computer. The Wiz proves to be quite a formidable opponent!

The third and most sophisticated game, Takeover, is a business-strategy game in which you control the fate of large corporations. Your goal is to become the most powerful business tycoon through clever wheeling and dealing in the stock market.

Each of the three games is explained in a separate chapter that describes loading instructions, the object of the game, playing instructions, and helpful

strategy hints. In addition, the chapter on Takeover, the most complex game, includes an expanded discussion of the technical aspects of that game. We suggest that you play each game several times before you read the chapters on game design. Numerous examples from the three games are used throughout the book in discussions of the game-design process, so you would find yourself quite lost unless you have played the games.

HOW TO USE THIS BOOK

Commodore 64 Advanced Game Design can be approached from three perspectives:

- Reading the book solely for the purpose of playing the three games.
- Using the book as a tutorial on how to program computer games. In this approach, you're satisfied with the game in your mind—what you want to learn is how to bring it to life in BASIC.
- Applying the book to learn the game-design process so that you can design, as well as code, your own computer games.

Whatever your perspective, we hope you enjoy playing the three games and learning the game-design process.

CHAPTER SUMMARIES

Chapters 2, 3, and 4 give instructions on playing the three games included in this book. Prime Time is discussed in Chapter 2, Word Square in Chapter 3, and Takeover in Chapter 4. We strongly suggest that you read these instructional chapters before you attempt to play any of the games!

Chapter 5, "Important Ingredients for a Good Game," discusses the qualities that make a good game and suggests ways of incorporating these qualities into game designs.

Chapter 6, "Game Design Process Step by Step," outlines a ten-step procedure for designing a game, from deciding on the game type through developing documentation.

Chapter 7, "Designing a Simulation Model," describes three types of simulations found in games:

real life competitive interactions, direct conflict, and statistically based simulations. Procedures for developing these simulations and examples of actual designs are given.

Chapter 8, "Artificial Intelligence: The Computer Plays," introduces the subject of artificial intelligence, discusses three categories of games that computers play, gives examples of the use of artificial intelligence in the games in this book, and describes the development of algorithms for the game of Dots and for a football simulation.

Chapter 9, "Programming Considerations," discusses computer RAM, choosing a language in which to program, input/output routines, program flexibility and structure, and execution speed.

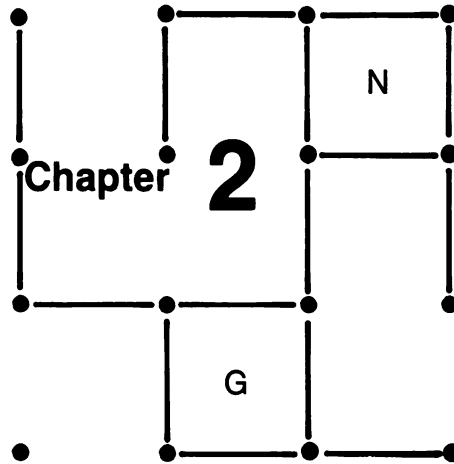
Chapter 10, "Implementing a Game Design: Professional Example Games," introduces the sam-

ple game programs included in the book and offers some suggestions for educational projects.

Chapters 11, 12, and 13 are the documentation chapters for the games Prime Time, Word Square, and Takeover. For each game, all variables, arrays, and subroutines are described, and flowcharts and the actual game listings are provided.

Chapter 14, "Making Money: Advice to Profiteers," offers suggestions based on our own years of experience on how to submit a game for publication.

Appendix A is a listing of the names and addresses of publishers who accept computer games from free-lance authors. Appendix B contains the conventions used in the program listings for this book.



Playing Prime Time

Have you ever wondered why a television program that you think is terrific gets taken off the air after its first season? Have you ever wondered why programs that you think are awful stay on the air year after year? The fact is, all television programs are pawns in what is commonly referred to as the ratings war.

We started thinking about television ratings about a year ago when we were called by the A. C. Nielsen Company and asked to be a "Nielsen family" for two weeks. Eager to see how the ratings process works, we readily agreed to participate.

While some families are issued instruments that attach to their television sets and keep track of what is being watched, our assignment was to keep a diary for each television set in the house. We were to record when each television was turned on and off, what was being watched, and who in the family was watching. In addition to these daily entries into the diary, we were asked to give our opinions, positive and negative, about specific television programs and about television in general.

Our participation made us feel that we did indeed have a voice in the process of program selection. Our viewing preferences, along with the preferences of thousands of other Americans participating in various surveys, were tabulated and analyzed by companies such as A. C. Nielsen. This type of survey can either make or break your favorite show.

A network executive's view of television programming is very different from a private viewer's point of view. Executives have to do a considerable amount of homework before the pilot for a new show goes on the air. Millions of dollars of the network's money are invested in the development process, and ideas are tested extensively in front of panels of viewers. You may think being a network executive is an exciting and glamorous job. At times it is, but people in this position are constantly under tremendous pressure to make wise programming decisions. Millions of dollars are being risked, and executives' jobs are at stake when ratings are low. You can experience the job of a network TV executive, with-

out all the high pressure that accompanies the job in real life, by playing the game Prime Time.

Prime Time is a game of strategy that pits the three major networks—ABC, CBS, and NBC—against each other in the ratings battle. The game can be played by one, two, or three participants. When only one or two people play, the computer plays the remaining networks.

LOADING INSTRUCTIONS

Whether you type in the Prime Time program or purchased the loader disk, the loading instructions are the same:

Type LOAD “PRIMETIME”,8 and press the RETURN key.

Type RUN and press RETURN at the READY prompt.

OBJECT OF THE GAME

In Prime Time, the players assume the roles of network television executives. The goal is to win the ratings war by having the lineup of programs with the highest yearly rating of the three networks.

OPTION PHASE

During the network-selection phase, you determine whether a human player or the computer will play each network. If a human is to play the network indicated on the screen, type a Y to indicate YES. Input an N to indicate that no human player will play this position; the computer will play it. The game has been designed so that one network has no advantage over the other two; therefore, the order in which each player takes its turn is random. A selections screen is displayed to summarize your selections. It shows whether a person or the computer is playing each of the three networks.

The fourth question “Enter # of weeks you wish to play,” gives you the opportunity to determine the length of the game. Although the computer accepts any number you enter, we suggest that a game of eight to ten weeks is optimal. After entering a number, press the RETURN key. The next screen displays your selections. If you press the Y key to

indicate that the selections are acceptable, the game begins. You can make changes in your responses to the four preceding questions if you indicate NO by pressing N.

BUYING SHOWS

Each network begins the game with \$126 million to spend on purchasing television programs. The game starts with a list of 18 fictitious television programs that the three networks take turns buying. The programs are listed in alphabetical order; the first letter of each show is used in making selections. The programs are also listed in descending order of *power*. This power rating shows how many millions of dollars a network must spend to acquire the program. The order in which the players choose shows for purchase is random.

If the computer is playing one or more networks and it is randomly selected to go first, then its selection is indicated in the owner column. When it is your network’s turn to choose a program (as indicated at the bottom left corner of the screen), you press the key that corresponds to the first letter (A through R) of your choice. The computer takes care of all calculations involving money transactions, and the balance for each network is indicated in the lower right of the screen during each turn.

As the selection process continues, each network’s choice is indicated, as are the computer’s choices. The column to the right of the screen shows you which programs have been chosen by which network so that you know which programs are still available. Of course, each program can be owned by only one network. If you attempt to purchase a show that has previously been selected, the computer instructs you to try again. This selection process continues until all 18 shows are purchased. In addition, there are three programs, Soapy, Trapping John M.D., and Unhappy Days, that the computer assigns randomly, one to each of the three networks. The prices paid for these three shows are determined by the amount of cash each network has on hand following the selection of its first six shows.

CHOOSING YOUR LINEUP

After each network acquires seven programs,

the game moves to the lineup phase. The screen shows the lineups for each network for week 1, beginning on Sunday and ending on Saturday. Your initial lineup is shown in the order in which you selected programs. At this time each network is assigned its own color. The seven shows you own are shown in the middle of the screen.

The chart at the top of the screen shows the days of the week from Sunday through Saturday. The column entitled VIEW shows how many million people watch television during the prime time slot on each night. Note that there are 75 million potential viewers on Sunday night, and the viewing audience drops off during the week to a low of 50 million viewers on Saturday night. The last three columns show the three networks and their current lineup for the week. Each program is indicated by its first letter only, and the number of power points (equal to price paid) for each program is also indicated.

When determining which of your seven shows you want to put on the air each night of the week, you should consider the following:

- the size of the viewing audience for that night,
- the power of each of your shows,
- the power ratings of the competing shows in the various time slots.

These are the primary factors that the computer uses to calculate the final ratings. Remember that to succeed in the ratings war, you have to be a ruthless executive!

The computer randomly chooses which of the three networks will present its weekly lineup first. To enter your weekly lineup, press the key that corresponds to the first letter of the program you want to show on Sunday night. Do the same for your other six shows in order (without spaces) from Monday through Saturday night. Then press RETURN. When you finish entering all seven shows, the computer gives you a chance to change your mind. If your lineup is not okay, you are given a chance to revise your lineup. The computer does not let you enter shows that you do not own. It also does not allow you to enter fewer than seven shows. If your

lineup as shown on the screen is exactly as you want it to be, then indicate that no changes are necessary by pressing RETURN.

There is really no clear-cut best strategy during the lineup phase. Obviously it would be wise to have your most powerful program shown on Sunday night when the viewing audience is larger than on any other night of the week. Chances are that your two network opponents will also feel as you do and put their most powerful programs on Sunday night, too. What happens when three powerful shows are up head-to-head against each other on the same night is that they tend to cancel each other out and all become less powerful, which lowers their position in the ratings.

Do not overlook the possibility of maneuvering your most powerful shows into the Friday and/or Saturday night slots. Because the size of the viewing audience is smaller on these two nights, your network opponents may keep their most powerful programs far away from these two nights. If you were to put powerful shows on these nights, the shows would have little competition from the other networks, and your shows then would do extremely well in the ratings. It is highly possible that a program shown on Saturday night could be #1 in the ratings!

After all three networks enter their weekly lineups, each network is asked, again in random order, if it wants to swap the positions of any two of its shows. Before deciding, look at the programs your shows are competing against on each night from the other two networks. If you decide to switch the positions of two shows, do so by entering the two numbers (1-7) that correspond to the nights on which these two programs appear.

The least enviable position to be in during the lineup phase is having to go first. The network that goes last has a big advantage because it can make moves based on what the other two networks have already done. Fortunately, Prime Time has been designed so that the order in which turns are taken is completely random. The consolation for going first is that you can swap two shows after all three networks have established their lineups. Naturally your opponents have the same opportunity to swap

two shows. Be sure to consider all of the possibilities before you swap. Try not to let the size of the viewing audience be too much of a determinant.

PRODUCTION AND PROMOTION

After all three networks have a chance to swap two shows, the next phase of the game begins. This phase concerns spending money on promotion and production for each of your shows. Promotion and production include such expenditures as advertising, special appearances by your stars, publicity, upgrading props and scenery, salary raises for your staff or stars, and so forth. Each network has \$100 million allotted per week to spend as it wishes on promotion and publicity, abbreviated on the screen as *P & P*. The money can be divided among your seven shows in any way you want. You can spend from \$0 to \$100 million on a show, but keep in mind that once you spend that \$100 million, you are out of funds for that week. If you do not allocate all of your money, the amount left over is spent on the seventh show in your lineup.

NEWS FLASHES

After the *P & P* phase of the game, there are three news flashes from Hollywood-type newspapers or gossip columnists. The three pieces of news are generated randomly, and the networks affected are also generated randomly. Just as in real life, sometimes the news is very good and sometimes the news is very bad. The ratings for the affected shows are helped or hurt accordingly.

RATINGS

Following the news-flash phase, the computer calculates the weekly ratings for each of the 21 programs in the game. The shows appear on a ratings chart, with the highest-rated show for that week at the top and the lowest-rated show at the bottom. Note that all of a network's shows appear on the list in the same color. The first column of numbers indicates the power rating. The numbers in the sec-

ond column indicate viewer share. At the bottom of the screen you see each network's weekly and yearly ratings rounded to two decimal places. Of course, after the first week the weekly and yearly ratings are identical.

When all players finish studying the ratings chart, continue the game by pressing any keyboard letter. Week 2 then begins. After the first week, you get a better feel for how your decisions affect the ratings war. The same procedure is followed throughout each week of the game.

END OF GAME

At the end of the last turn of the game, the final ratings are calculated. The network with the highest yearly rating is declared the winner.

HELPFUL HINTS

The following tips should help raise your scores in Prime Time:

- Keep a careful eye on the maneuverings of your competitors so that you can make counter-moves.

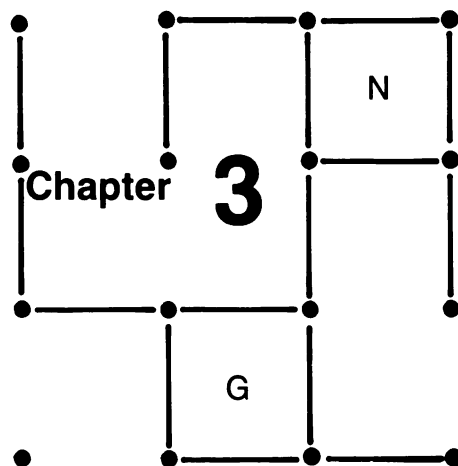
- Try to position your highest-rated shows in the time slots that have the least competition from the other networks and that have the highest possible viewer audience.

- Do not always do the obvious, particularly during the production and promotion phase. Well-placed spending can elevate a low-rated show considerably higher in the ratings.

- Allocate your \$100 million wisely each turn. If you give too much to one show, your other shows may suffer.

For further clarification, a detailed explanation of how the Prime Time ratings are calculated is given in Chapter 7, "Designing a Simulation Model."

Now that you have had a glimpse of how a computer can play along as a game participant, it is time to meet a really smart character—the "Wiz." Word Square is explained next.



Playing Word Square

Word Square is a positively addictive word game in which you compete against the computer and against up to four human players. If you enjoy word games or crossword puzzles, Word Square is for you. The object of the game is to use random letters to form five words that score more points than the words formed by your opponents. The computer, whose name is Wiz, may prove to be your most challenging opponent in Word Square, as it has a vocabulary of more than 3,500 words! The computer's words have all been validated by the *Official Scrabble Dictionary*.

LOADING INSTRUCTIONS

Whether you type in the Word Square program or have purchased the loader disk, the loading instructions are the same:

Type LOAD "WORD SQUARE",8 and press the RETURN key.

Type RUN and press RETURN at the ready prompt.

OPTION PHASE

First, you see the Word Square title page. Next comes the Option Phase, which consists of a series of questions that you answer to determine the nature of the specific game you want to play. You can play a solitaire game against the computer Wiz, or the game can be played by up to five human players along with the Wiz. Press RETURN after you enter the number of human players who are participating. Each human player is identified by three initials, and the order in which you enter each player's initials is the order in which each player takes a turn. Press RETURN after entering each set of three initials. Next you can determine the length of the game by deciding how many turns the game will consist of and the number of minutes per turn. A turn can last as long as 10 minutes or as short as one minute. The shorter the turn, the more challenging the game. Again, press RETURN after each entry.

Following the last entry, a summary of your selections is shown. If they are acceptable, press the

Y key to indicate YES. If the selections are not okay and you wish to make changes, press the N key to indicate NO, and the Option Phase is repeated.

LETTER VALUES

The next screen shows you the point values of each letter of the alphabet, as shown in the following chart:

Letter Values

10—Q Z
8—J X
5—K
4—F H V W Y
3—B C M P
2—D G
1—A E I L N O R S T U

The point values are identical to those used in the game of Scrabble. When you form words it is obviously to your advantage to be able to use those letters that have high point values so you outscore your opponents. For example, when was the last time you had a quaff at a quay (a deep drink at a wharf)? Have you ever seen a zoril on a zax (African Skunk on a roofing tool)? Or have you ever made a quip about a quin (a sarcastic remark about a variety of pectin)? Also, never try to jape a jager (trick a huntsman), but always put on a jupe (loose jacket) when you go out for jurel (an edible fish). You may not be familiar with these words yet, but you can be sure that the Wiz knows them, and your human opponents might, too! So learn to spot those high-value letters and use them to form words whenever possible.

FORMING WORDS

Each player needs paper and pencil to play. (The Wiz does not have to write anything down because it has a terrific memory!) The upper left-hand corner of the screen is a clock that counts to the time limit in each turn. Next to the counting clock is the time limit per turn that you entered during the Option Phase. The upper right-hand corner of the screen displays which turn you are currently play-

ing and the total number of turns you determined for the game.

Press any key to being the first turn. The seven boxes across the top of the board fill with randomly generated letters as the turn begins. The likelihood of a particular letter being generated by the computer is equal to the approximate likelihood of that letter appearing in the English language.

At this point it may be helpful to give an example. Let us say you are presented with the following seven letters across the top of the board:

S L A I M J R

The point values for these letters would be:

S=1 L=1 A=1 I=1 M=3 J=8 R=1

Of the seven given letters, five are valued at only 1 point, while the M and the J are more valuable. When forming words, therefore, it would be wise to try to use and J and the M as often as possible. The S is also a valuable letter. Although it is valued at only one point, it can be used repeatedly to make plurals of your words.

The center of the screen displays the five-by-five playing board square in which your words are formed. Five randomly generated letters fill in the square on a diagonal from the upper left box to the lower right box of the square. Looking across the top row of boxes, you form a word with the letter that appears in the first box plus letters from the seven boxes at the top of the screen. Just as in Scrabble, the word cannot be a proper noun or a foreign word, and it must consist of at least two letters.

The letter in the second box of the second row must be used as the second letter of your next word, and again you must use the letters from the seven boxes across the top of the screen to form your word. Likewise in the third, fourth, and fifth rows, the given letters must be used respectively as the third, fourth, and fifth letters of your words. Each of the letters from the seven boxes at the top can be used only once in any row but can be repeated for use in other rows.

After time expires, a tone sounds and the letters from the seven boxes across the top of the screen disappear. It is now time for the players to enter their words and for the computer to calculate the scores. The player whose initials were typed in first is asked to type in his or her word from the first row. After the word is typed in, it is shown as typed. You are asked if the word is valid or invalid according to the rules established previously. If there is a question concerning the validity of a particular word, a dictionary should be consulted. If the players agree that the typed word is valid, type V. If the word is invalid, type I. If you attempt to use letters that are not given initially in the seven boxes, or if you attempt to repeat a letter, the computer displays INVALID WORD, and lets you correct a possible typing mistake.

If you do not use a given letter in its correct position, the computer displays an error message. If a typing error occurs while a word is being entered, press the T key (for TRY AGAIN), and you can retype the word without penalty. If a player cannot come up with a word for a given row, he or she simply presses the RETURN key and the computer moves on to the next row's entry. The process is repeated for the other four words for that player. Then the computer requests words from the second player, the third player, and so forth.

If all the players finish forming their words and do not want to wait until the clock counts to the established time limit, press the left-arrow key on your Commodore 64 to begin the word-inputting process.

SCORING

The computer calculates your score for each turn by adding the point values for each letter used in all five words and multiplying the total by the number of words (one to five) each player used. After the words for each human player are entered, the Wiz displays its five words in the square. Sometimes the Wiz is not able to come up with words for all five rows. These words represent the best the computer can find within the extent of its vocabulary. Can you beat the amazing Wiz?

The box at the left of the screen shows the score for that turn for the Wiz and for each of the human players. The total score at the right of the screen shows the cumulative score for the Wiz and for each player. Turn 2 will begin after you hit the key.

END OF GAME

At the conclusion of the game, the Final Summary screen displays the final cumulative score for the Wiz and for each player. Each player is given a rating that ranges from NOVICE to WIZARD, depending on the score achieved relative to the Wiz's score. You are asked if you want to play another game. If you press the Y key to indicate YES, then the Option Phase begins again. If you press the N key to indicate NO, then the program ends.

RATING YOUR PERFORMANCE

The following calculation is used to determine your rating:

$$\begin{aligned} & (\text{Player's point score} / \text{Wiz's point score})^3 = R \\ \text{Your ratings number (RA)} &= \text{INT} (R \times 6 - \text{time} \\ & \quad \text{per turn in minutes} \times 2 / 10) \end{aligned}$$

The rating number is between 0 and 5.

RA	Rating
5	Wizard
4	Genius
3	Master
2	Expert
1	Student
0	Novice

Part of the beauty of Word Square is that you can play the game at various levels of difficulty, which means that even young children can join in the fun. Also, you can play by yourself—just you against the Wiz! (That rascal has on many occasions kept us awake into the wee hours of the night!)

Unlike Scrabble, Word Square contains no element of luck. You cannot complain that you drew

lousy letters, as each participant in the game works with the same set of letters. (Nor will all of your letter tiles accidentally get knocked off the board!)

HELPFUL HINTS

The following tips will help you score more points:

■ Use the high-value letters whenever possible. With the Q and the Z worth 10 points each and the J and the X worth 8 points each, it is advantageous to familiarize yourself with words containing these letters.

■ Keep in mind that a short word with high-value letters scores higher than a five-letter word with low-value letters.

■ Do not spend too much of your precious time thinking about one row while ignoring the other four rows.

■ Try not to leave any rows without a word because any word is better than no word. The scoring method uses both the letter values and the number of words created.

■ Take advantage of an S appearing among

your given letters to form plurals. Look for letter combinations that may be helpful in forming other word endings, such as ED or ING.

■ Keep your eye on the counting clock so that you do not run out of time.

SAMPLE GAME

Yes, the Wiz is quite smart, but it can be beaten, as the following sample game shows. This game consists of two turns, each lasting 3 minutes. The participants are the Wiz and a person named You. Figure 3-1 shows how the playing board looks at the beginning of turn 1. Note the 10-point Z and the 8-point X.

The person named You came up with three words in the three-minute time limit. The word ID scored 3 points, DEAD scored 6 points, and DAZE scored 14 points, which equals a letter-value total of 23. To compute the total score for You's first turn, the number of words created is multiplied by the total point value: 3 × 23 = 69 points, which is recorded in the box at the left of the screen. See Fig. 3-2.

The Wiz did better than You did this turn. It

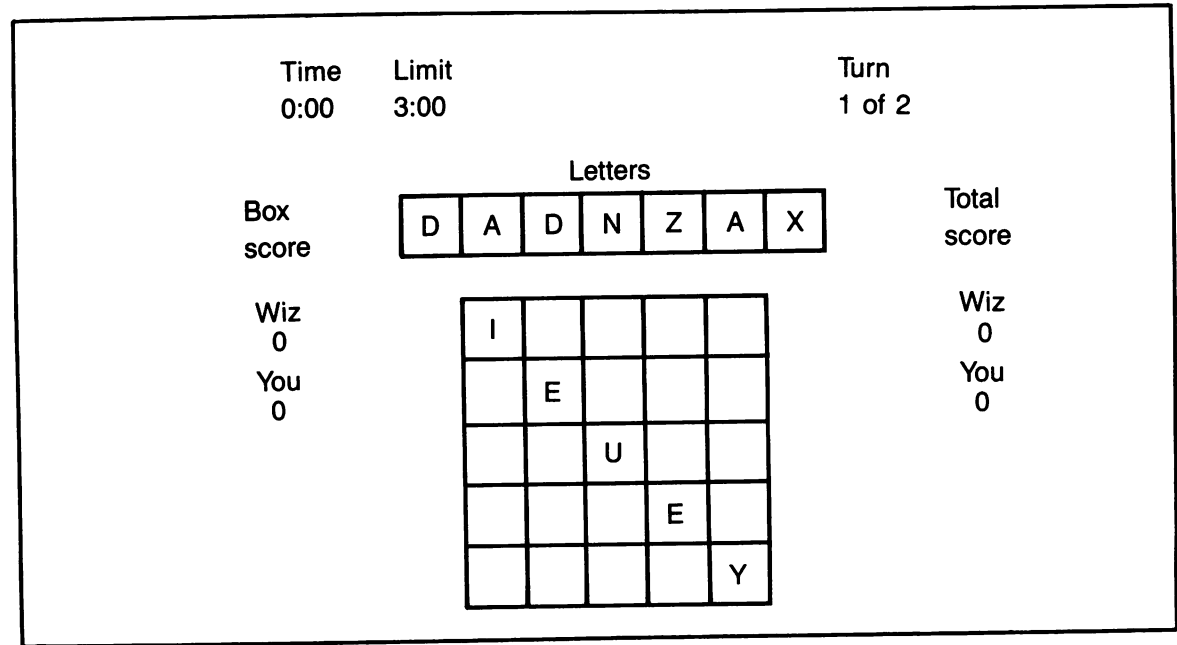


Fig. 3-1. The Word Square game at the beginning of turn 1.

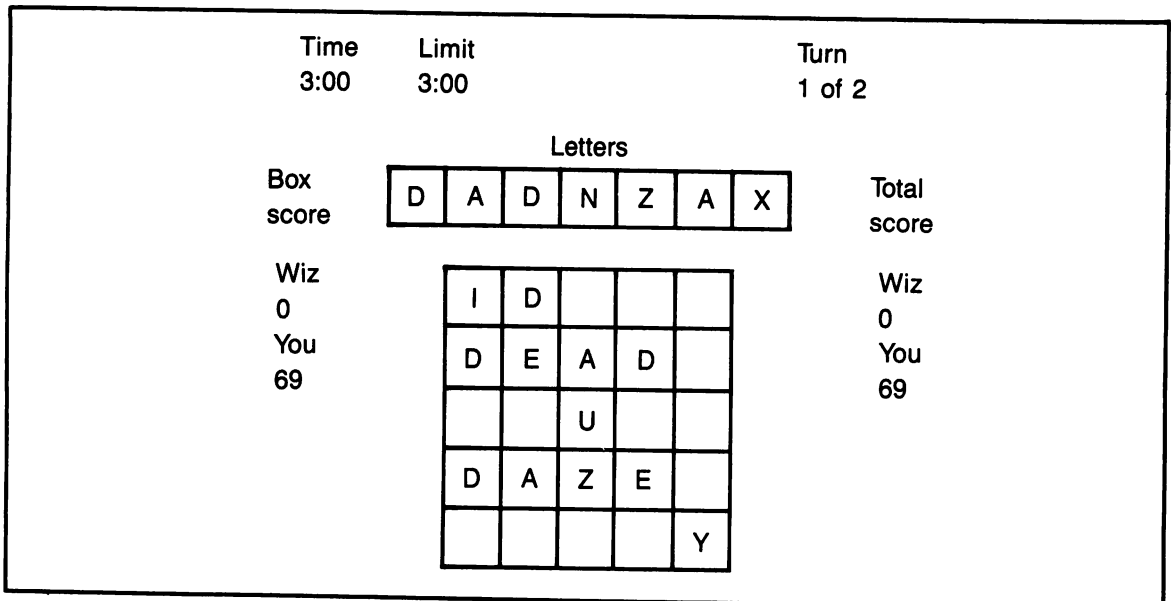


Fig. 3-2. The Word Square game after the player's turn.

scored 3 points for ID, 13 points for ZED, and 14 points for ADZE, which equals a letter-value total of 30. Since the Wiz formed three words, its total score for the turn is figured as follows: $3 \times 30 = 90$ points. See Fig. 3-3.

"Not bad," you say, but what are an adze and

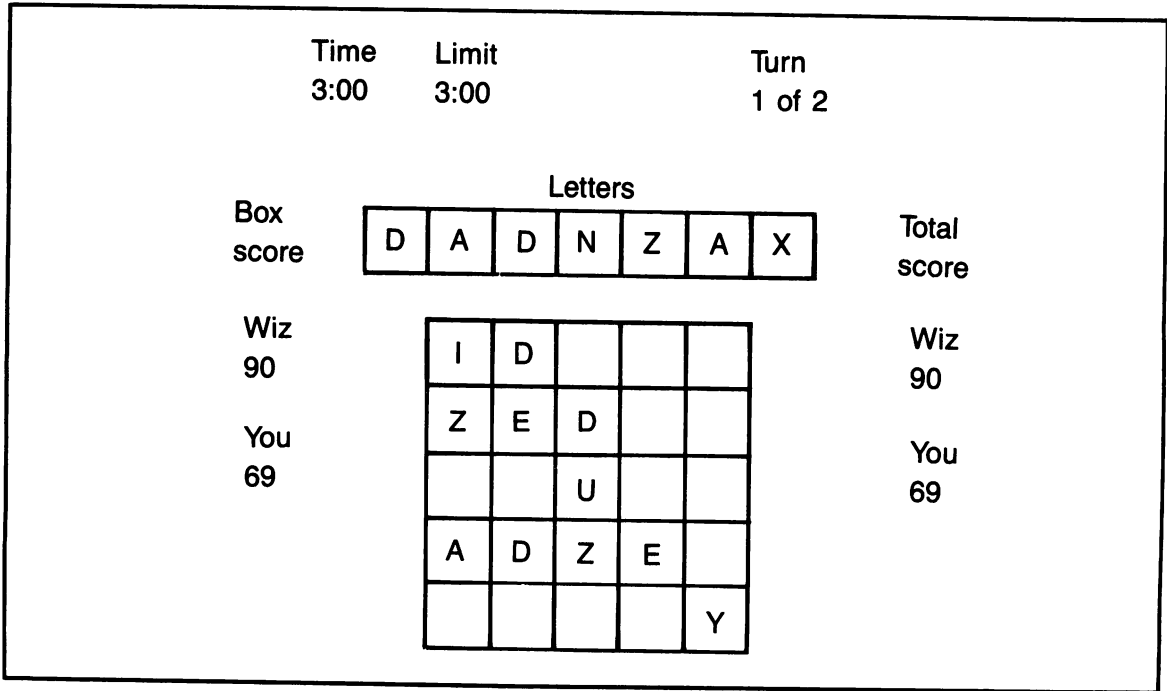


Fig. 3-3. The Word Square game after the computer's turn.

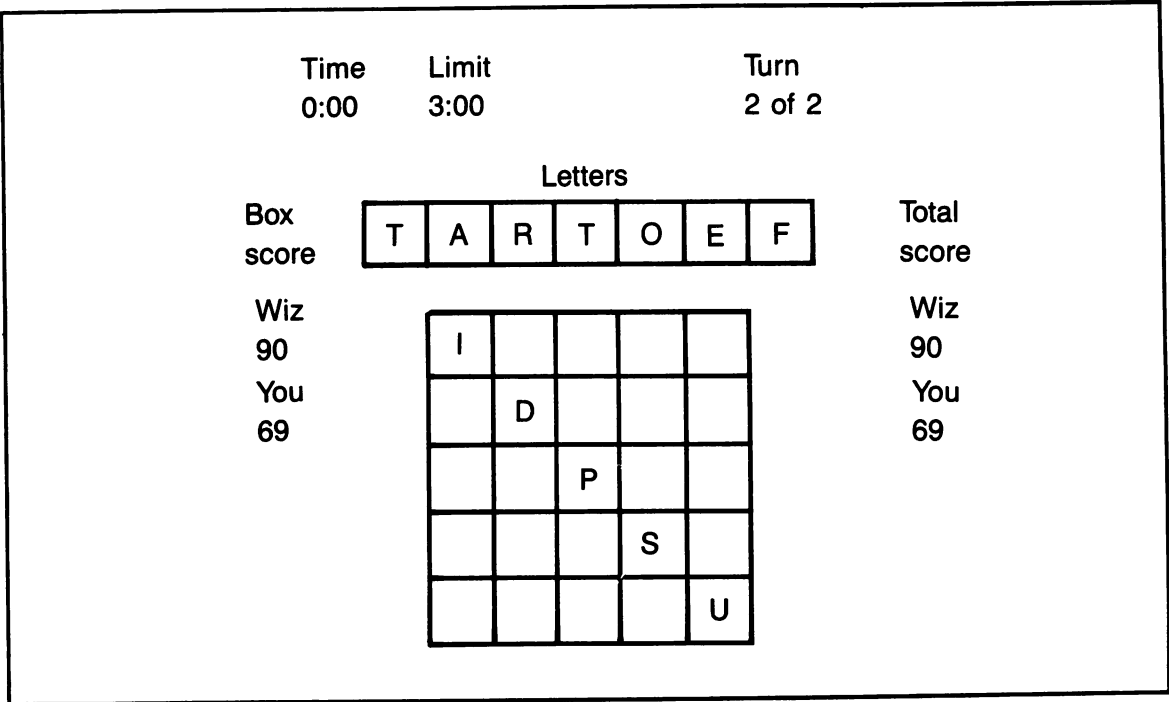


Fig. 3-4. The Word Square game at the beginning of turn 2.

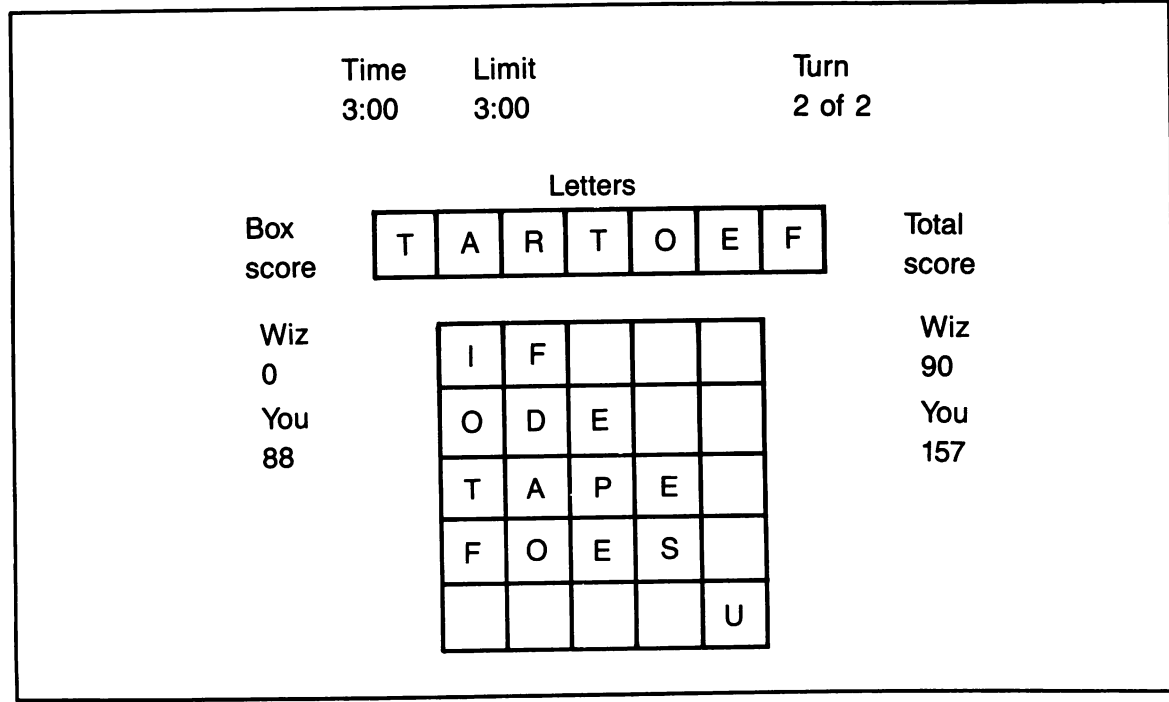


Fig. 3-5. The Word Square game after the player's second turn.

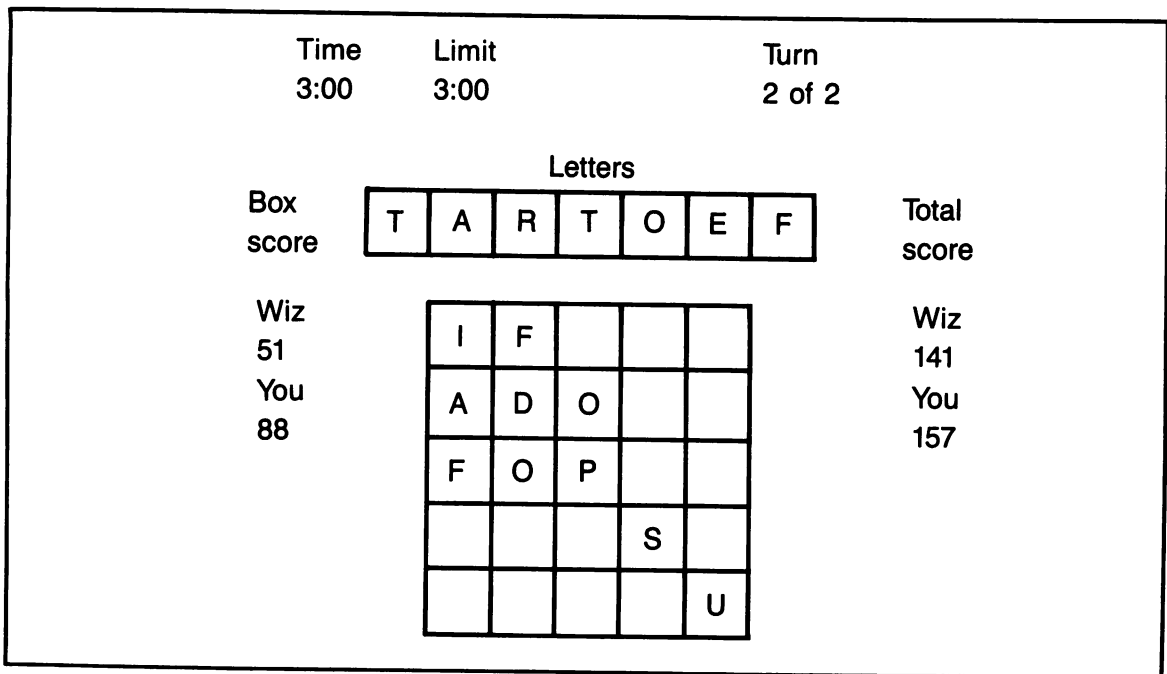


Fig. 3-6. The Word Square game after the computer's second turn.

a zed? According to Webster, an adze is a sharp-edged wood-cutting tool, and zed is the word for the letter Z. We told you the Wiz knew a lot of words!

In the second turn of this sample game, there are no particularly high-value letters to work with. See Fig. 3-4.

You did very well this turn in coming up with four words. You used the f twice, which is the highest-value letter on the board (four points). See Fig. 3-5.

The word IF scored 5 points, ODE scored 4 points, TAPE scored 6 points, and FOES scored 7 points, which equals a letter-value total of 22. This total multiplied by 4 gives You a total point score of 88 points for turn 2. See Fig. 3-6. Well done!

The Wiz did not do nearly as well as You did this turn. The Wiz scored 5 points for IF, 4 points for ADO, and 8 points for FOP, which equals a letter-value total of 17 points. Unfortunately, the Wiz was

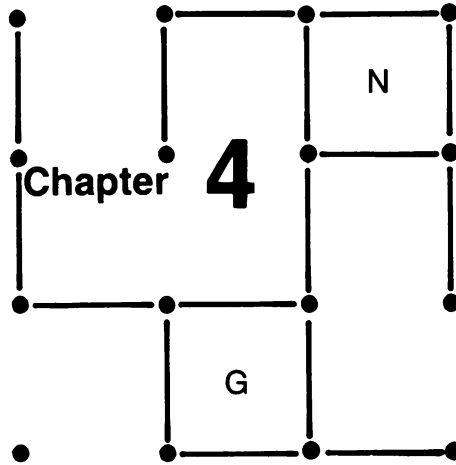
not able to come up with a word for the fourth row. The reason is that the Wiz does not know how to add an S to the end of a three-letter word to make it plural. The Wiz is smart, but not that smart! This is an example of a situation in which a human has an advantage over the Wiz because a person knows how to make plural words. The Wiz's total point score for turn 2 is $3 \times 17 = 51$ points. FOP, by the way, is a noun meaning a dandy.

Figure 3-6 shows the final total scores for the two-turn game on the right side of the screen.

WIZ 141
YOU 157

For beating the Wiz, You get a WIZARD rating. However, try beating the Wiz in a 10-turn game . . .

Are you ready to do some wheeling and dealing in the world of high finance? Takeover is next, so read on.



Playing Takeover

Undoubtedly you have seen the word *takeover* in the newspaper headlines a lot lately. This is because we are currently experiencing a merger and acquisition boom in the business world. The current acquisition game began in 1982 with Bendix, Martin Marietta, and Allied Corporation making headlines daily. Their debacle started a new era in corporate deal making. Corporate executives, reading the business section of the newspaper, see all these big deals taking place, and they want to get into the act and buy somebody, too! Thus, many little corporations are “eaten up” by the already-big corporations, and this makes the latter even more powerful.

Would you like to play the part of the corporate executive involved in high-powered wheeling and dealing in this acquisition game? You get this opportunity in the game *Takeover* (without actually risking your shirt!). You need not have a college degree in finance or a seat on the New York Stock Exchange to enjoy playing *Takeover*.

Takeover is a business-strategy game that can be played by 1 to 10 players. The object of the game is to amass the largest net worth so you can become

the most powerful business tycoon. Each player takes on the role of an executive of a major corporation. The number of major corporations in the game is the same as the number of players. All the remaining corporations become minor corporations. Through a series of investments, stock purchases, stock sales, and takeovers of minor corporations, each major corporation attempts to outmaneuver its opponents to gain power and fortune and win the game.

LOADING INSTRUCTIONS

Whether you type in the *Takeover* program or purchased the loader disk, the loading instructions are the same:

Type LOAD “TAKEOVER”,8 and press RETURN.
Type RUN and press RETURN at the READY prompt.

OBJECT OF THE GAME

The object of the game is to have the highest

personal or owner net worth at the end of the game. Owner net worth is equal to the total worth of the corporation times the percentage of the total outstanding shares owned by the player:

OWNER NET WORTH = (Value of Stock in Other Corporations + Cash + Investments - Debt) * Blocks (10,000 shares) of Stock You Own / (Blocks of Stock Owned by other Players + Blocks of Stock on the Open Market)

OPTION PHASE

First you are asked if you want to load a saved game (Y,N). If you press the Y key, then you are asked for the name of a saved game. This game is loaded and resumes at the beginning of the turn following the turn at which it was saved. If you type a wrong game name, an I/O error occurs. You are advised of the error number, and you can retype the correct name of your saved game.

If you want to begin a new game, press N. You are asked for the options for the new game. There are two game options: the number of players and the number of game turns. You can select from 1 to 10 players and from 1 to any number of turns. To get the full enjoyment from the game and proper interaction between players, we suggest a game of 10-20 turns. Be forewarned, however, that this is quite a lengthy endeavor. You may have to save the game so you can play it in more than one sitting. It might be helpful to play short games of 5-8 turns in the beginning while you learn the rules and strategy.

There is no provision in Takeover for the computer to manage one of the major corporations, so it is not an active participant. The role of the computer is to handle the minor corporations, to simulate the stock market, and to handle all transactions.

After you make your selections, the options as selected are listed and the computer responds with the prompt:

"Are these selections OK (Y/N)."

If you want to make a change, press N and the

option selection phase is repeated. If you press Y, then a new set of corporations is created and a new game begins starting with the first turn.

SETTING UP CORPORATIONS

Following the Option Phase, the setting up of corporations takes place. This is done only once, at the beginning of the game. Player 1 is assigned to play ATT, player 2 becomes BIC, player 3 becomes CBS, and so forth. The values for each corporation are determined randomly by the computer within rules that are discussed later. Each major corporation, the number of which is determined by the number of game participants, begins the game of Takeover with \$50 million in total corporate net worth.

Whenever a new game is started, the values for the corporation parameters are established before the main display is first printed on the screen. These parameters are determined by two methods; the method used depends on whether the corporation being established is a major or minor corporation. Acronyms, such as ROI (return on investment), are clarified later in this chapter.

Major corporations are set up under the following rules:

- The total worth of the corporation, CASH + IVST - DEBT, is set equal to \$50 million.

- CASH is determined randomly by the computer at between \$10 and \$30 million.

- IVST (long-term investment) is determined randomly at between \$20 and \$60 million.

- DEBT is set based on CASH and IVST so that the total worth equals \$50 million.

- ROI (return on investment) is set equal to 10 percent.

- There are one million outstanding shares with 60 percent held by the owner and 40 percent held on the open market.

Minor corporations are set up under the following rules:

- CASH is determined randomly between \$1 and \$11 million.

■ IVST is determined randomly between \$1 and \$11 million.

■ DEBT is set randomly to $(0 \text{ to } .5) * (\text{CASH} + \text{IVST})$.

■ ROI is determined randomly between 10 and 50 percent.

■ There are 500,000 shares with 100 percent held on the open market.

MAIN DISPLAY

At the top of the display is the current market price in dollars for a share of stock in each corporation. These base stock prices are determined by a corporation's net worth and the number of shares outstanding on the open market. This base price is the amount you pay or receive for one share of stock. When you buy or sell stock, the price you pay or receive is an adjustment from the base price. For example, when buying stock, you pay more than the open-market base price. The more you buy, the more expensive the stock is. When you sell, the more you try to sell, the less of a price per share you receive.

The corporations are represented by a three-letter symbol as listed below:

ATT	American Telephone Transmission
BIC	Bethlehem Iron Corporation
CBS	Celebrity Broadcasting System
DEC	Digital Electric Company
EPC	Eastman Photo Company
FAC	Federal Acceptance Corporation
GAC	General Auto Corporation
HBA	Home Builders of America
IBM	International Banking Machines
JJI	Johnson & Jones Incorporated
KCC	Kaiser Coal Company
LOC	Louisiana Oil Company
MMM	Minnesota Metal Manufacturing
NBC	Nashua Broadcasting Corporation
OEC	Overland Express Company
PPR	P & P Railroad
QOF	Quince Orchard Foods
RAP	Royal American Petroleum
SOC	Sexxon Oil Corporation
TUS	Transportation U.S.

Note that each corporation has a unique first letter. This single letter must be used to refer to that corporation in all game commands.

At the center of the main display are the corporate balance sheets. At the left is the balance sheet of the corporation owned by the current active player. At the right, which is initially blank, the balance sheet of any corporation can be displayed for examination. (See Check Sheet command.) The corporate balance sheet contains information on the following parameters that make up the corporation:

■ ROI is the return on investment (IVST) listed as a percentage for the corporation. At the end of each turn, an amount equal to this percentage of a corporation's investment (IVST) total is added to CASH.

■ CASH represents the total liquid assets of a corporation in millions of dollars. CASH can be used to buy stock, pay off debt, and make additional investments in the corporation.

■ IVST represents the total amount of long-term assets of the corporation in millions of dollars. It is the quantity that is multiplied by the ROI to obtain the per-turn income for the corporation.

■ DEBT represents the total corporate debt in millions of dollars. At the end of each turn an amount equal to the product of the interest rate (IR) times DEBT is subtracted from a corporation's CASH: $\text{CASH} - (\text{IR} * \text{DEBT})$.

■ OWN is the total number of blocks (BLKS) of stock that the owner/player of this corporation currently owns. A block (BLK) of stock is equal to 10,000 shares. Only the major corporations (See Setting Up Corporations) have an owner. OWN is equal to zero for minor corporations.

■ PLY is the total number of blocks of stock in the corporation that are owned by the players of the game. For major corporations, the owners' shares are shown under OWN and are not part of this total.

■ MRK is the total number of blocks of stock in the corporation available for purchase on the open market. (See the section "Market-Price Determination" and the Purchase Stock and Sell Stock commands.)

Below MRK is the total worth of the corporation in millions of dollars. The total worth of the corporation is calculated by multiplying the price per share times the sum of OWN + PLY + MRK. Below IVST in the initially blank area is a list of stock in other corporations that this corporation owns. This list shows the three-letter symbol of the corporation and the number of blocks owned.

At the bottom of the display screen is the command menu (which is explained in the command-menu section), the current turn number, and the interest rate (IR). The interest rate is the percentage of DEBT that is deducted from CASH at the end of each turn.

PLAY SEQUENCE

The game lasts the number of turns specified in the Options Phase, with each turn following this sequence:

- Market prices are calculated.
- Interest rate is set.
- Each of the players uses the command menu to enter commands for the turn.
- End-of-turn processing is performed.
- Corporate and total worth are displayed, and the player is given the opportunity to save the game.

A player can have only one transaction with each other corporation within a given turn.

MARKET-PRICE DETERMINATION

The price of an individual share of stock is determined for each corporation by the division of the total worth of the corporation by the total number of outstanding shares (OWN + PLY + MRK). It is this price that is shown at the top of the main display. The price paid for a share of stock in a transaction on the open market is an adjusted value of this base price. The adjustment factor uses the following formula to increase the price for buying transactions and decrease the price for selling transactions:

$$\text{Adjustment factor (ADJ)} = 1 + \frac{\text{BLKs being bought or sold}}{\text{MRK}}$$

$$\begin{aligned} \text{Price paid per share} &= \text{Base price} / \text{ADJ for selling} \\ &= \text{Base price} * \text{ADJ for buying} \end{aligned}$$

INTEREST RATE CALCULATION

The interest rate is set at 12 percent during game initialization. At the beginning of each turn the interest rate is allowed to fluctuate randomly about the previous turns' interest-rate value. The interest rate determined by this process is limited so that the minimum interest rate calculated for the turn is greater than or equal to 5 percent and the maximum is less than or equal to 20 percent.

COMMAND MENU

Each of the commands listed in the command menu is accessed by pressing the key corresponding to the first letter of the desired command. You are then prompted through a series of further commands as required. Numeric input, such as blocks of stock, prices, and dollar amounts, are entered by typing the number and pressing RETURN. Commands are invoked by simply pressing the indicated key (RETURN is not required). When executing a command that involves a stock transaction, you refer to the stock by pressing the first letter of the corporation whose ownership represents the stock. When executing a command that involves another player in the game, you refer to this player by pressing the first letter of the major corporation that this player is managing. If you want to terminate a command sequence initiated from the main command menu and that is now in progress, you may do so by pressing the **£** key in place of commands or by entering a 0 value for numerical input. This termination returns you to the main command menu.

The following commands are available for use:

B—Borrow Money. A player may attempt to borrow money from the bank. After the player enters the loan amount in millions of dollars, the computer analyzes the corporation's debt/equity ratio and either approves or rejects the loan applica-

tion. If the loan application is approved, the amount of money requested is added to CASH and to DEBT. This money is available immediately for use in transactions during the current turn.

C—Check Sheet. Press C and the first letter of a corporation to display that corporation's balance sheet on the middle right side of the main display.

D—Debt Payoff. This command allows a player to use CASH to reduce the amount of DEBT. Any amount up to the amount of CASH available may be specified.

E—End Turn. This command ends the current player's turn, and control passes to the next player. After the E key is pressed, the computer checks whether the player really wants to end the turn; this prevents the premature end of a turn if you inadvertently press the E key. Press Y (for YES) to pass control to the next player. Pressing N to indicate NO returns you to the command menu.

I—Investment. This command allows a player to add any amount up to the current value of CASH to long-term investment (IVST). The amount specified is subtracted from CASH and added to IVST. Note that once CASH is converted to IVST, it remains there. There is no command available to convert IVST to CASH.

P—Purchase Stock. This command allows a player to purchase stock in a corporation. You may purchase stock from another player or the open market. You may not purchase stock in a major corporation from the player who owns that corporation. This player must use the Sell Stock command to offer the stock for sale. The only limits on the number of shares you may purchase are the number available and the amount of CASH you have to use for the purchase.

The price you pay for a share of stock in an open-market purchase is affected by the number of shares you want to purchase and the number of shares available, as explained in the Market-Price Determination section. You may check this price by using the Quote command. If you are buying stock from another player, the price paid can be any amount that is mutually agreeable to the parties making the transaction. Stock purchased in corpora-

tions other than your own are displayed on the balance sheet below IVST. Stock purchased in your own corporation is added to OWN and is not displayed.

The Purchase Stock command requires a number of subcommands that may be confusing until you have played Takeover for awhile. We hope the following example situation, which involves the purchase by Bethlehem Iron Corporation (BIC) of stock in Eastman Photo Company (EPC) from American Telephone Transmission (ATT), clarifies how commands are entered to execute this transaction. In this example ATT and BIC are major corporations and EPC is a minor corporation. We assume that ATT purchased 10 blocks of EPC stock on the open market during previous turns of the game.

1. BIC presses the P key to purchase stock.
Computer prompts: "Purchasing Stock.
Press key to select corporation/STOCK"
2. BIC presses E to select the stock to purchase.
Computer prompts: "From whom do you wish to purchase
(M)arket (A)nother player"
3. BIC presses A since ATT is managed by another player.
Computer prompts: "Purchasing stock from another player
Press key to select corporation/STOCK"
4. BIC presses A to select ATT as the corporation from which shares are being purchased.
Computer prompts: "Purchasing stock in EPC from ATT
Enter blks to buy, price per block"
5. BIC enters 10,20 (to buy 10 blocks at \$20 per block).
Computer prompts: "Transaction as entered
Buy 10 blk(s) of EPC for 20
ATT is this acceptable (Y,N)"
6. ATT presses Y to indicate YES.
Computer prompts: "Accepted"

Other purchase options as well as the Sell Stock command follow similar command sequences.

Q—Price Quote. This command displays the price per share and the total amount of a buy or sell transaction made on the open market. It adjusts the base price with the appropriate factor to account for the number of shares in the transaction and the number of shares available on the open market.

R—ROI. This command displays the percentage return on investment for each corporation in the center of the screen. This display is replaced with the normal balance sheet command as soon as another command is executed.

S—Sell Stock. This command allows you to sell stock in a corporation. You may sell stock either to another player or the open market. You may sell any number of shares up to the number you currently own. *Short selling*, the selling of more stock than you own, is not allowed. The amount you receive from a sale on the open market is affected by the number of shares you sell and the number of shares already available on the open market, as explained in the Market-Price Determination section. You may check the price per share and the total amount of a transaction by using the Q command. If you are selling stock to another player, the price paid can be any amount that is mutually agreeable to the two parties involved in the transaction. Stock sold in corporations other than your own are subtracted from the amount displayed under IVST. Stock sold in your own corporation is subtracted from the amount OWN.

T—Takeover. With this command you attempt to take over a minor corporation. You may not take over a major corporation. For a takeover attempt to be successful, you must own more than 50 percent of the total outstanding shares (PLY + MRK). If a takeover is successful, then the computer merges the two corporations in the following manner:

1. The CASH, IVST, and DEBT values of the two corporations are added.

2. A new ROI is calculated as the weighted average of the two corporations. The IVST value of each corporation is used as the weighing factor.

$$\text{New ROI} = \text{IVST (major corp)} * \text{ROI (major corp)}$$

$$+ \text{IVST (minor corp)} * \text{ROI (minor corp)}$$

3. A liquidation value is calculated for the acquired corporation. Additional shares in the new merged corporation are issued and distributed to previous owners of stock, including the open market, in the corporation that has been taken over.

All commands may be repeated as many times as you want. However, you are allowed only 1 buy-or sell-stock transaction per turn. For example, if you purchase 10 blocks of ATT in turn #5, you may not purchase more ATT or sell any ATT stock until turn #6. You may continue to buy or sell stock in BIC or any other corporation.

END-OF-TURN PROCESSING

After all players finish their turns, the computer performs the following end-of-turn processing:

1. The investment return for each corporation is calculated by multiplying the ROI by IVST. This product is added to CASH.

2. The amount of interest due is calculated by multiplying IR by DEBT. This product is subtracted from CASH. If the amount of interest due is greater than the amount of CASH, then the amount by which CASH is exceeded is deducted from IVST.

3. The amount of CASH remaining for minor corporations is transferred to IVST.

SAVING THE GAME

At the end of each turn, you are given the opportunity to save the game for play at another time. If you select this option, you are asked to give the saved game a file name. You must not specify a name that is the same as a file currently on the disk. You use this new file name to retrieve the saved game when you resume play at a later date.

END OF GAME

After the number of turns specified in the Option Phase is completed, an End-of-Game Summary is displayed. The player who has the highest owner worth is the winner. You are given the opportunity to play another game.

STRATEGY AND HELPFUL HINTS

There is no best strategy for playing Takeover since your decisions are strongly affected by the number of turns in the game and by the decisions made by other players in the game. However, there are a number of facts you should keep in mind when you prepare the business plan for your corporation:

- The ROI (return on investment) for the minor corporations is considerably higher than that for the initial major corporations. This provides an incentive to take over the smaller companies so that your investment dollars yield a better return.

- Since the cost of stock increases with the amount purchased, it is usually wise not to attempt to obtain majority interest in a corporation in one turn. The manner in which you build up the necessary shares should take into account the number of turns remaining in the game and the ROI

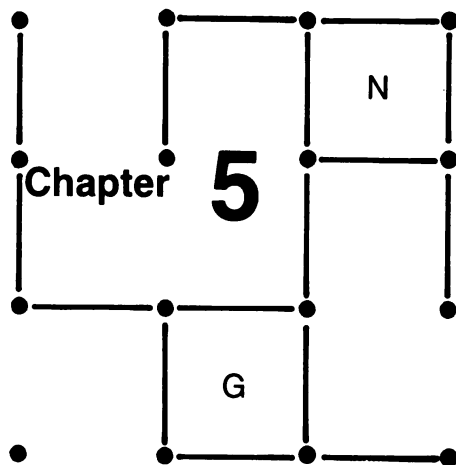
and IVST (long-term investment) amounts of the corporation.

- The best corporations to take over are those that have the highest ROI and IVST values. Your objective should be to maximize the amount of increase in your own corporation's ROI. You must carefully follow and anticipate the other players' transactions with the best minor corporations so that you can set your business plan accordingly.

- Your CASH position should be kept at a minimum since you receive no return from CASH.

- Excess CASH can be used to decrease DEBT if IR (interest rate) is greater than ROI, or to increase IVST if ROI is greater than IR.

Now that you know how to play the three games in this book, you are better able to follow the examples taken from these games to illustrate ideas in the upcoming chapters on game design.



Ingredients for a Good Game

Welcome to the exciting world of microcomputer game design! If you want to do more than just play the three games presented in this book, then read on. Perhaps your curiosity is aroused and you would like to know just what goes on in the microcomputer game-design process, or maybe you are interested in designing a professional game for submission to a publisher. Now that you are familiar with Prime Time, Word Square, and Takeover, we would like to delve into the game-design processes that were used to develop these games. For this chapter and the remaining chapters, we use examples from these three games when they illustrate the game-design lesson we are trying to convey.

Have you ever sat back after playing a game that you particularly liked and thought about what makes it such a good game? If not, an enjoyable exercise would be to make a list of your 10 favorite games. These need not necessarily be computer games. After you make your list, attempt to characterize those features that are common to all, or at least to most, of these games. These features

comprise the most important ingredients for making a good game for you. In the following pages we explain our most important ingredients and give some examples of games with these qualities. We also describe some of the ways these qualities can be incorporated into your game designs.

A good game should follow these principles:

■ *Be simple to understand yet difficult to master.* The best example of such a game is chess. Most people can learn the rules for playing chess and the moves of all of the pieces in a single evening. Yet people have spent lifetimes studying the game and have yet to come up with a clear-cut best method of play. The move combinations and strategies possible in chess are so numerous that no one can decide on the single best order of moves to make.

Of course you cannot expect to design a game with the depth and beauty of chess your first try, but this does not mean that you cannot design a game with many of the same qualities. For example, word games, such as Word Square, are certain-

ly easy to learn, and we doubt that you will deal crushing defeats to your able computer opponent the first few times you play.

If you plan to design a simulation game like Takeover, however, you will realize that the simple-to-understand quality starts to conflict with the degree of realism that you build into your design. You must walk a fine line in this area and make trade-offs between complexity, playability, and realism. If you are undecided about particular trade-offs, it is usually a good practice to simplify the design and sacrifice some realism. Keep in mind, however, that if you are designing the game for your own enjoyment, and, if the utmost in realism is one of your key ingredients, then by all means build into your design as much realism as you can within the constraints of your time and your computer system.

■ *Reward good play and punish poor play.* It is important that games reward good play and punish poor play. To achieve this objective, care should be taken not to have too large a luck factor. Luck is part of life and most simulations include it, but when you implement randomness in your design, you must be careful that it does not introduce such a large uncertainty factor that it makes strategic planning a waste of time. For example, Takeover requires quite a bit of long-term planning. You must plan your budgets, decide on borrowing and investment strategies, and research the minor corporations so you can spot prime take-over candidates.

There are two types of random elements built into Takeover. The first is a pure random-number generation of the interest rate for each turn. This certainly affects play, but it should not overwhelm proper play on the part of any player. For that matter, a player should understand the randomness of the interest rate and devise a business plan that is not too susceptible to major swings in the interest rate. A second type of luck element in Takeover involves the effects of buying and selling decisions made by the other players. This is the best sort of luck element, since the other players' actions are not truly random and might be anticipated in each player's business plan. This human element is a very important ingredient in the design of Takeover, since it defeats what we call *one-dimensional strategies*.

As you are planning the design of a game and you attempt to develop it so that proper play is rewarded and poor play is punished, you have to be careful that you do not design a single-best-play strategy. If you do, your game will become quite boring as soon as this play strategy is discovered. Random elements are the easiest way to defeat a one-dimensional strategy, but again we urge caution and recommend that randomness be caused by the nature of the game or through player interactions and not be achieved through a proliferation of random-number generators.

■ *Provide a balanced contest between participants.* The values of the most important game parameters should interact properly. The difficulty in achieving play balance varies widely, depending on the nature of the game and the level of experience of the players. Chess, for example, is perfectly symmetrical, with both players having identical armies. This type of game is fairly balanced by design, and special rules or scoring are not required to balance play. But you still might ask about the advantage that the player of the white pieces has in chess or how a balanced contest can be achieved between players of vastly different experience levels.

Historical simulations, such as war games, offer a much greater challenge for the game designer attempting to achieve play balance. For example, After Pearl, a game that we designed for SUPERware, simulates the war in the Pacific following the Japanese attack on Pearl Harbor. Since this is a historical simulation, we were somewhat limited by the realities of history. The Japanese and American navies had vastly different force strengths throughout the war. The Japanese were considerably more powerful than the Americans at the beginning of the war, but the Americans became much more powerful by the end of the war as the industrial might of the United States was brought to bear.

There are a variety of methods you could choose to achieve balance in such a game. First, you might ignore the need for balance by simply developing an arbitrary scoring scheme for the game and establish victory conditions for certain final scores. For example, a score greater than 50

might signify an American victory and a score of less than 50 might signify a Japanese victory. This is certainly the easiest way to achieve play balance, but in most cases the results are the least satisfactory. Another method for designing play balance is to develop a scoring scheme for the game based on the historical importance of game objectives and then adjust the game probabilities. For example, you could adjust the probability that a Japanese plane scores a hit against an American surface ship until equal scores are achieved by players of approximately equal experience and skill. This is better than the method of arbitrary scoring, but usually does not provide the most enjoyable game.

We found that the best method for achieving play balance in *After Pearl* was to adjust the game rules, the strategic positions of forces, and the scoring values for objectives. These adjustments were made to be significant within the historical framework of the game. For example, the American navy is allowed to take the first shot in each battle phase. This rule tends to increase the effectiveness of the American navy so a more balanced contest is achieved. However, it also has a historical basis, since the American navy had broken the Japanese communications code earlier in the war and often had early warnings of Japanese attacks and troop movements. The scoring system of the game is designed to place historical significance on the values of objective naval bases, but it is also balanced with the rules so that equal players achieve approximately equal scores by the end of the game. We find it much more satisfying for a player to win the game by outscoring the opponent. Victory conditions that require certain point-total differences seem to be a little too nebulous to understand.

■ *Be challenging, but not impossible.* All good games must be winnable, and it must be clear to the player that the game is winnable. Great care must be taken in designing this aspect of your game. You must be careful not to push the challenge of your game to either extreme. A game that is too easy to win quickly bores the players and is discarded in the junk pile. Likewise, if a game is too difficult, players might give up long before they appreciate the depth and quality of your design.

Most difficult games can maintain the players' interest even if the players continue to lose. If players can clearly see why they lose, what errors they made during the game, and what they can do to improve play in the next game, they will be anxious to start play the next time they boot the game into the computer. The best games are those that allow players to continually improve their performance through experience, ultimately to the point where they achieve victory at least some of the time.

In solitaire games, even the best players should not be able to beat the computer every time. *Word Square* is an example of such a game. In multiplayer games, such as *Prime Time*, that are designed primarily for competition between human players, the quality of the computer player need not be the mark of excellence against which human play is measured. The computer needs only to be able to play a credible game that will not adversely affect the result between the human players through foolish moves. This makes designing the computer artificial-intelligence routines for such a game much simpler than programming a computer to play, for example, checkers or chess.

■ *Satisfy the player's ego.* Many successful games, particularly those of the arcade variety, appeal strongly to the players' egos. The arcade hit *Defender* is an excellent example of this technique, with its list of highest-scoring players prominently and boldly displaying the top player. Although such considerations need not be an inherent part of your game design, it is a good practice to add something similar to a list of high-scoring players. Such appeals to players' egos increase player satisfaction and make them want to continue playing to improve their standing on the list. We believe this single feature causes many people to continue to play some arcade games long after their interest in the game for the game's sake has subsided.

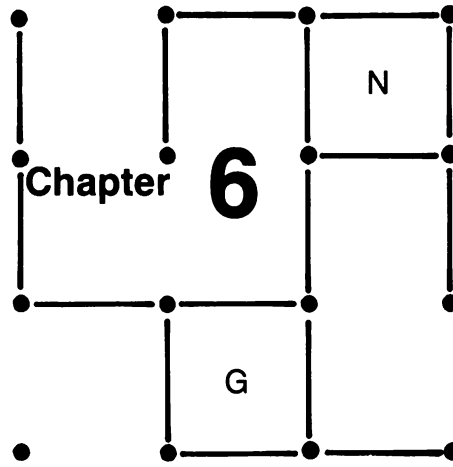
■ *Avoid being affected by player-computer interface.* The design of the input/output structure for your computer game is of the utmost importance and must be carefully designed along the lines described in the chapter on programming considerations. You must strive to make the interface between the human and the machine the simplest means for

the players to interact with each other and with the computer. Shy away from fancy, eye-catching schemes and keep the I/O structure as simple as possible. Your goal is to make the player-computer interface transparent to the player. This interface must not distract the players' attention so that they can put their full effort into playing the game in the computer and not playing with the computer.

The objectives of any game-design project are based on these six most important ingredients. You should constantly evaluate your efforts against these features through all phases of your game-design project from the conceptual stage through the final play-testing stage. As you work on your game, ask play testers for their honest comments on it. At-

tempt to ascertain what they like and dislike about your entire game and determine which particular features in your game appeal to them the most. Evaluate how these comments relate to the list of ingredients of a good game. This evaluation will indicate how successfully you have incorporated the six ingredients into your game.

It may be that you will have to make some modifications or additions to your design. To prevent the need for major revisions in your design and to make the necessary revisions as easy to implement as possible, you need to follow a well-structured plan for your game design. In the next chapter we present a step-by-step game-design process that provides such a structured plan.



Step-by-Step Game-Design Process

The emphasis in this chapter is on the actual design of the game. Programming considerations are covered in detail in later chapters. In this chapter we walk you through the design process step by step. This process gives you a good plan to follow as you design your first microcomputer game.

Whether you are designing a game for your own use or for profit, the game cannot be done in a haphazard fashion. The complexities involved in combining game design with computer programming require that a structured approach be taken. A 10-step procedure for designing a game is given in the following pages. We strongly recommend that you follow this procedure or some similar structured approach when you design a game.

STEP 1: DECIDE ON GAME TYPE

So you want to design a game! An excellent way to start is to choose the type of game you want to write. Your choice should be based on the programming skills necessary to implement designs of each type as well as on your knowledge of the subject

matter around which your game will be centered. You should realize that certain topics lend themselves to certain types of game formats. For the purpose of discussion in this book, computer games are broken into four types: arcade, abstract strategy, adventure/word games, and simulations.

Arcade Games

Computer arcade games are similar to those you can play at your local market or video parlor after you plunk your quarter into the slot. These games are heavily graphics oriented and are mainly a test of a player's eye-hand coordination and reaction time. These games are almost always played against the machine, with the primary objective being to increase your score. The depth and strategic planning required for these games is very limited since it is not convenient to supply a player with a long or complicated set of rules for play. Nor does the machine's owner want you playing a time-intensive game for your one quarter. Although these same considerations do not apply to the home computer,

most arcade-type programs written for home computers have been written in an attempt to copy or emulate video-machine versions. The primary reason for this is that software publishers want to capitalize on the popularity and name recognition of such popular titles as PAC-MAN and DONKEY KONG.

Arcade games are not a good type of game to start with when you are learning game design and programming because this type of game has complex graphic operations, and, therefore, requires a very experienced programmer. BASIC is much too slow to handle the graphic manipulations required; most arcade games are written in machine code. In addition, it is much more difficult to create an original design within the arcade framework. It is particularly difficult to build in strategic planning and interactions between multiple human participants. Lacking this capacity for planning and player interaction, arcade games tend to hold a player's interest for a shorter period of time.

Abstract Strategy Games

Abstract strategy games involve strategic planning and may involve conflicts that tend to be abstract in nature. Abstract strategy games make only limited attempts, at best, at simulating historical occurrences or human situations. Examples of abstract strategy games are backgammon, checkers, and chess.

There is considerable variation in the difficulty in designing and programming these types of games. For example, a game like backgammon confronts the player with many situations that require precise calculation, but not a lot of long-term strategic planning. Developing the computer artificial-intelligence algorithms for this type of game is more straightforward than for games such as checkers and chess. Also, the dice add a significant random element to the game, so a computer can be programmed to play a very respectable game of backgammon.

Programming a computer to play checkers is more difficult than programming it to play backgammon since checker moves require a greater level of long-term planning. Also, checkers does not

lend itself to a precise calculation of moves. On the other hand, checkers algorithms are considerably simpler than chess-playing algorithms since the positioning and combinations possible on a chess board far outnumber those on a checker board. Chess is considered the ultimate game of strategy by many people, and the programming of computers to play chess has been used for many years to develop algorithms in the field of artificial intelligence. While it is not recommended that you start your game-design career with a chess-playing program, a simpler, more calculative abstract strategy game might be a good place to start. This is particularly true if your main interests lie in the area of computer artificial intelligence. The game of Dots, which is discussed in the chapter on artificial intelligence, is a good example of a very simple abstract strategy game.

Adventure/Word Games

Adventure games are really puzzles. The designer creates an environment or world into which the player is placed. The player is given a defined goal, such as finding and collecting treasures. The player usually interacts with this artificially created world by typing two-word noun and verb combinations at the keyboard. Since, by necessity, the computer's vocabulary is limited, a big part of adventure gaming is determining the noun-verb combinations that allow you to progress toward your goal. A well-designed adventure leads the player to the proper word combinations with clues that are discovered as the designer's universe is explored. The puzzles presented in a good adventure should be logically solvable from the information presented to the player and should not be tricky.

Adventure programs can be written in BASIC if you can sacrifice the extra running time required. Designing a good adventure can be fun, but it is a complex and fairly lengthy process. The adventure-game design process is not described in this book because the procedures involved are so lengthy and are somewhat different from much of what is described in this volume. Adventure-game design could be the subject of an entire book.

Word games have been somewhat arbitrarily

grouped with the adventure games. Word games require considerable text processing, just as adventure games do, but the design of a word game is usually much less complex than the design of an adventure game. If you enjoy playing word games, they probably are the best type of games to start designing. They are graphically simple, and the programming implementation can be accomplished by a less experienced programmer. Word games can be quite varied, as they allow the novice game designer to be original and creative. The Word Square program included in this book offers an excellent springboard for other similar word programs. You can use the Word Square word vocabulary, add to it, or modify it for countless other games.

Simulations

Simulation games attempt to simulate historical occurrences, such as the Battle of the Bulge, or attempt to let the player assume a role, such as that of a cutthroat president of a large corporation. Prime Time and Takeover are examples of simulations that let the player enter the business world and allow for multiple player interaction. War games, such as After Pearl are heavily conflict-oriented and usually simulate major historical battles. Our game Baseball's Best is an example of a statistical game that uses the records of real baseball players to simulate the game of professional baseball.

In addition to word games, business-type simulations, such as Prime Time and Takeover, are an excellent type to choose for your first game-design project, particularly if you are knowledgeable in the area you are simulating. War games and statistical games require considerable background research and may require more time to develop than the average person is willing to spend. For this reason, we recommend that you first try a game along the lines of Prime Time or Takeover. After you finish such a project, the design of a war game or a statistical simulation will be easier and require less time and effort.

Cross Ties

There are numerous combinations of the

previously mentioned types of games. For example, we wrote a game called Panzer War, which is a strategic simulation of the war on the Eastern Front during World War II that relies on arcade phases for the tactical battle sequences. Properly written games that combine the best elements of the various types of games can be the most fun and challenging to play. Unfortunately, such games are the most difficult to design and program. You should not attempt such a combination design until you have designed and programmed at least one (and preferably more) game from each type you want to combine in your final design.

STEP 2: CHOOSE A TOPIC

After you decide on the game type, you should next choose the topic for the game. For example, if you choose to write a simulation game, you might then decide to write a statistical type of game. You might choose to simulate a sport, and then decide on a particular sport, such as baseball. The most important consideration in choosing your topic is to select an area to simulate that you know and enjoy. After all, you are trying to have fun! If you are not trying to produce a game for commercial sale, you need not be concerned with those topics that currently have strong market appeal.

STEP 3: DEVELOP A GENERAL SIMULATION MODEL

After you know the type of game you want to write and you have decided on a topic, you should develop the general approach for simulating the subject. This approach, which depends on the game type and topic, is described in detail in Chapter 7, Designing a Simulation Model. You need not develop the exact mathematical algorithms for the simulation at this point, but you should outline the most important factors relating to the subject and jot down a few approaches to simulating them. You should also outline how the factors will interact and the importance that each should have. You must start to consider programming methods and estimate whether your proposed model approaches are feasible within the execution time and RAM

constraints of your computer system.

STEP 4: OUTLINE PROGRAM FLOW

By this point you should have a general idea about how you are going to design your game. You have decided on the key factors and have a generalized model for simulating their effect and interaction. Next you need to outline the general flow of your program. This need not take the form of a formal flowchart, although it could. We have found that the best approach is to devise a large, pseudo-flowchart for the entire program. On the chart you should mix descriptions of logical program flows, mathematical formulas, lists of subroutines that might be required, and actual lines of code. This program flow plan should indicate how the principal objectives and goals of the game are achieved. It should contain programming notes and considerations, with emphasis on the I/O structure of the program. The plan should show estimates of the values of the main program parameters that will produce a balanced and challenging contest. You should build flexibility into the flow plan so that changes to parameter values and program flow can be accomplished without drastically altering the program.

STEP 5: DEVELOP DETAILED SIMULATION MODELS

The next step is to transform the notes from the program flow-chart into detailed algorithms for simulating each factor. A good approach is to code these algorithms into subroutines that will later be used in your programs. In this way you can discover programming difficulties that might not have been apparent previously. Also, you can avoid major programming modifications that accompany the discovery that a proposed model cannot be implemented for one reason or another. It is sometimes necessary and prudent to sacrifice some realism in your game so you can solve the problems of program implementation. This process is described in detail in Chapter 7, Designing a Simulation Model. In this chapter, examples are given of simple models for a variety of game topics.

STEP 6: WRITE A ROUGH PROGRAM

It is finally time to transform your program plan into computer code and start typing in your program. The details of this process are discussed in Chapter 9, Programming Considerations. At this point, you should not be concerned with fancy options or the computer artificial-intelligence algorithms. You need to develop a working program that will allow you to decide if your program plan makes sense and if it produces an enjoyable and challenging game.

STEP 7: PLAY TEST THE ROUGH PROGRAM

The play testing of your rough program is one of the most important and time-consuming steps in the game-design process. You have many objectives to achieve during this step, the most important of which are:

- Modify and refine the rules of play, if necessary, to improve playability, balance, and game challenge.
- Adjust major program parameters, such as starting position, net worths, or per-turn revenues, to achieve the desired balance of importance between factors.
- Debug your code.
- Jot down proposed improvements in program flow, I/O, or presentation, that will improve playability and appearance.
- Learn the strategy for play. Your notes on proper strategy are needed to form the basis of the computer's artificial-intelligence routines.

STEP 8: DEVELOP COMPUTER ARTIFICIAL INTELLIGENCE MODELS

At this stage, you transform your strategy notes into algorithms for computer play, as described in Chapter 8, Artificial Intelligence: The Computer Plays. You must consider the computer RAM available and the execution time required for your routines. You need to convert your algorithms into computer-code subroutines that, if your program flow plan is done properly, will fit neatly into your rough program.

STEP 9: PLAY TEST THE COMPLETED GAME

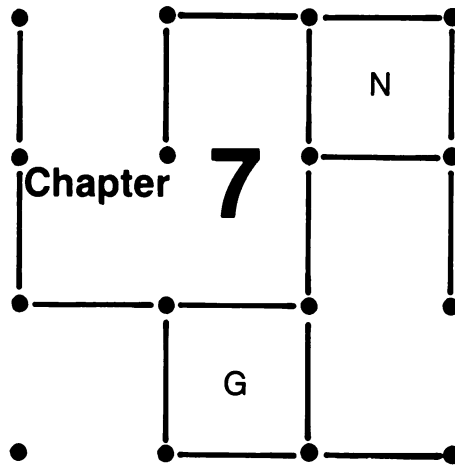
At this point, you add the programming enhancements planned in Step 7 and the computer intelligence routines from Step 8 to play test the complete game. Unfortunately, you will find that you are far from achieving a finished product; problems and new ideas for improvements continue to pop up during play testing. You should enlist as many of your relatives and friends as you can find to play test your game so you get varied opinions and suggestions for improvement. Consider using play testers of both sexes and of various ages.

A particularly powerful tool for turning up problems with your game or program is the blind test, in which players totally unfamiliar with your game play it without your being present. Be sure to provide your play testers with some written instructions to follow. The feedback from such tests can be very enlightening.

STEP 10: DOCUMENTATION

Naturally, writing documentation goes on throughout the entire design process. It is a good idea to write out some simple rules of play for your game. This is a necessity if you want to have your game blind tested. Program documentation serves as an invaluable aid as you plan improvements in your game design or as you generate ideas for other new games. Keep all your notes, particularly the program flow plan, that will be essential if you want to modify your game later.

If you think that the microcomputer game-design process we just described is long and involved, you are right! Should you decide to develop your own game, you must be prepared to spend countless hours at your computer keyboard. Two of the steps in this process, designing a simulation model and developing artificial-intelligence routines, are especially complicated, and we devote the next two chapters to explaining further the intricacies involved in these steps.



Designing a Simulation Model

One of the most difficult and time-consuming steps in the game-design process is the development of the simulation model. In developing the model, you must constantly be aware of the limitations set by your computer RAM and by the execution speed of the algorithm you choose for your model. Such limitations compel you to make frequent trade-offs between realism and playability. Even though BASIC is probably the best language for you to use, one area in which BASIC is inferior to languages such as FORTH and assembly is execution speed. If you do choose BASIC for writing your program, you must be extremely careful when you design your simulation algorithms not to develop equation systems that take BASIC forever to solve.

This section describes three types of simulations:

1. Real-life competitive interactions without actual conflict, such as business-strategy games.
2. Direct conflict, as in war games.
3. Statistically based simulations, such as a

computer simulation of baseball.

We describe some simple procedures used to develop simulations in these types of games and give some practical examples of actual designs.

REAL-LIFE COMPETITIVE

The game Takeover is a good example of a real-life competitive simulation. There is no combat or direct conflict in the game, but there is considerable competition and interaction between the players. Let us assume that we are at step 3 of the design process of the game Takeover. We have decided to design a simulation-type game and have settled on management of a big corporation as our topic. We have specialized the topic somewhat by deciding to make the takeover of minor corporations by the monopolistic major corporate powers be the central theme for the game. We'll make a list of the principal factors in this environment that we want to simulate. Then we'll look at the algorithms used in the game Takeover to model these factors.

Principal Factors

The following aspects of Takeover are to be simulated:

- Buying and selling stock on open market.
- Representation of major and minor corporations.
- Independent investment strategies of minor corporations.
- Market interest rate.

Buying and Selling Stock on the Open Market

Clearly, since our simulation must include a considerable number of stock transactions, we must develop a model to simulate the stock exchange. To simulate the market, we need to determine the buying and selling price for shares in each corporation. We want to build in as best we can the market forces that affect stock prices. Since our central theme involves the takeover of small corporations by the corporate giants, we need to show the strong effect that the supply of shares for sale has on the price of a stock. Players should pay a premium price to obtain the last few shares necessary for a takeover and should receive a low price if they have to unload large blocks of stock on the market following an unsuccessful takeover attempt.

Takeover uses a two-step approach to model this situation. First, a base or market price of the stock is calculated. This is the price that you would have to pay for a single share of stock. Next, an adjustment factor is applied to the base price to account for the number of shares being bought or sold. The base price is determined from the net worth of the corporation according to the following formula:

$$\text{BASE PRICE} = (\text{CASH} + \text{INVESTMENT} - \text{DEBT} + \text{STOCK VALUE OF OTHER CORPORATIONS OWNED}) / \text{TOTAL OUTSTANDING SHARES}$$

The adjustment factor is calculated as follows:

$$\text{ADJ} = 1 + (\# \text{ of SHARED BOUGHT or SOLD} / \# \text{ of SHARES AVAILABLE on OPEN MARKET})$$

The open market does not contain shares owned by any of the contestants in the game. The adjustment factor is then applied as follows to determine the price players pay for each share in a transaction.

$$\text{BUYING STOCK: PRICE} = \text{BASE PRICE} * \text{ADJ}$$

$$\text{SELLING STOCK: PRICE} = \text{BASE PRICE} / \text{ADJ}$$

Thus, through some simple formulas we are able to simulate the market forces we outlined. Admittedly, one could dispute the realism or accuracy of such a model. We agree it is somewhat arbitrary, and countless other factors could be added. However, we believe this is a good example of a case where realism can be sacrificed for playability since the model clearly provides for the correct price-movement trends. The play-testing phases of the design process must be used to assure that these price movements are properly balanced against the other major factors in the game. They should strongly affect play but not overwhelm other planning factors that we want to highlight in the design of Takeover.

Representation of Major and Minor Corporations

We need to represent the corporations so that the major decisions that a corporate director must make are reasonably simulated. However, we must attempt to simplify this representation as much as possible so that the game will not become overly complex.

The model implemented in Takeover uses four main variables, CASH, INVESTMENT, DEBT, and ROI (return on investment), together with stock ownership to simulate a corporation's balance sheet. CASH is used to represent all the liquid investments of the corporation. INVESTMENT represents longer-term investments such as those for plants and equipment. DEBT equals all the liabilities of the corporation; the corporation is charged a fee for DEBT in each turn. The fee amount depends on the interest rate. ROI is the percentage of return on the INVESTMENT; a player receives this amount in each turn.

With just these simple aggregate quantities, we simulate a great number of the difficult questions that a corporate manager must face. Should I borrow from the bank to finance INVESTMENT expansion? Should I raise money by selling corporate stock? Should I expand with only internally generated funds? Should I expand my operations into other fields by attempting to take over another corporation? Should I speculate in a variety of stocks, hoping the price will rise because another corporation is attempting a takeover? Through questions such as these, you can see that many combinations are possible within the few parameters chosen. By choosing the aggregate parameters as we did, we are forced to ignore corporate decisions such as management of accounts receivable, labor, internal investment, research and development, and so forth. As a designer, you face difficult trade-offs in the design of the simulation model. The great fun in designing your own game is that you get to choose what is modeled explicitly and what is abstracted.

Once the major corporate parameters were set for Takeover, a method was chosen to model these parameters for each corporation. To do this, the following ground rules were established:

1. The major corporations should have different balance sheets, but the overall corporate values should be the same.
2. The minor corporations should be much smaller than the major corporations.
3. The minor corporations need not have the same values.
4. There should be an incentive for the major corporations to take over the minor corporations.

The following equations were developed to satisfy these ground rules.

For Major Corporations:

1. CASH = Random from \$20 - \$30 million.
2. INVESTMENT = Random from \$40 - \$60 million.
3. DEBT = CASH + INVESTMENT - \$50 million.
4. ROI = 10 percent.

5. OWNER STOCK = 600,000 shares.
6. OPEN MARKET = 400,000 shares.

For Minor Corporations:

1. CASH = Random from \$1 - \$11 million.
2. INVESTMENTS = Random from \$1 - \$11 million.
3. DEBT = (CASH + INVESTMENT) * Random (0 - 1/2).
4. ROI = Random 10 - 50 percent.
5. OWNER STOCK = 0.
6. OPEN MARKET = 500,000 shares.

If you examine equations 1 and 2 for major corporations, you find that the major corporations are allowed to have a random makeup, yet are forced by equation 3 to have equal value. Thus we satisfy ground rule 1. The constants in the equations for the minor corporations are set so that smaller values are obtained for CASH, INVESTMENTS, and DEBT, which fulfills ground rule 2. Equation 3 for the minor corporations fulfills ground rule 3 by not forcing the minor corporations to the same value as the major corporations. Finally, to fulfill ground rule 4 for providing a takeover incentive, we set the ROI of the major corporations to 10 percent and give the minor corporations an ROI that is random and always greater than the major corporations. This simulates the greater returns on investment that emerging growth companies usually have and at the same time causes the major corporations to want to acquire these companies to increase the profitability of their investment dollars.

Independent Investment Strategy of Minor Corporations

Since the minor corporations are not being directed by a human player, their actions must be controlled by the computer. The minor corporations are not intended to compete with the major corporations in the stock acquisition and takeover of the minor corporations, and, therefore, are not allowed to purchase stock in other companies. This greatly simplifies the simulation of their investment actions.

Next, we assume that the minor corporations will not borrow and finance all expansion with only internally generated cash flow. This makes the entire simulation of minor corporation investment strategy come down to the single equation.

**NEW INVESTMENT=CASH AVAILABLE
AFTER PAYMENT OF INTEREST ON DEBT.**

Such a simple equation allows rapid calculation of minor corporation strategy in BASIC. A much more complicated model could be developed, but such a model would cause a considerable time delay between turns while BASIC crunched away at these equations. This one simple equation produces the desired effect, since the minor corporations will continually grow in value in proportion to their ROI and will be affected by initial CASH and DEBT positions and by the prevailing interest rate. This simple model is justified by the fact that the players are not playing against the computer and the minor corporations for game victory. If you wanted the computer to direct one of the major corporations, a considerably more complicated model would have to be developed.

Market Interest Rate

Finally, we must determine the prevailing market interest rate the banks will charge the corporations for their debt. The market interest rate is an extremely complicated value to determine and is affected by a vast number of competing and interrelated economic factors. Within the scope of the game, the actions of these corporations should have very little effect on the value of market interest rates. Therefore, the assumption that market interest is random is a good one. We assume the market interest rate starts at 12 percent and varies each turn with equal probability by as much as 4 percent up or down, with a maximum value of 20 percent and a minimum value of 5 percent.

PRIME TIME RATINGS MODEL

Prime Time is another example of a real-life competitive game. The principal simulation model

in this game is the model for the calculation of the show ratings for the week. The ground rules used for the development of this model in Prime Time are as follows:

1. A show's rating should be based on its direct audience appeal or power rating, the number of viewers available in its time slot, and the power ratings of competing shows that are placed in the same time slot.

2. The network executive should have the ability to increase an individual show's power rating by spending extra money on production and promotion for the show.

3. The power rating of a show should be dependent on the show's rating performance during the previous week.

4. The power rating of a show should be affected by random events beyond the control of the network executive.

The algorithm used to calculate the rating of shows in Prime Time is as follows:

RATINGS=VIEWERS IN TIME SLOT * (ADJUSTED POWER OF SHOW/SUM OF ADJUSTED POWER OF ALL SHOWS IN TIME SLOT)

ADJUSTED POWER=POWER RATING BASED ON PREVIOUS POSITION IN RATINGS LIST (30-RATING #) + INCREASE IN POWER RATING DUE TO SPENDING ON PRODUCTION AND PROMOTION (\$\$\$/3) + THE EFFECT OF RANDOM EVENTS (+/- 0-10)

These simple equations fulfill all the ground rules we set for the model. Since the rating is a fraction of the total viewers in a time slot, with the fraction determined by the ratios of the show's individual rating to the total rating of the shows in the time slot, ground rule 1 is clearly satisfied. Ground rule 2 is adhered to because the network executive can increase the show's adjusted power rating directly by spending additional money on production and promotion for that show. The base power rating for the show is equal to 30 minus its

position in last week's ratings list; this satisfies ground rule 3. Finally, ground rule 4 is fulfilled because random events occur each time, with each random event increasing or decreasing the show's power rating anywhere from 0 to 10 points.

This simple model reasonably simulates the two principal decisions a network executive must make:

1. In what time slot do I place each of my available shows?

2. How much money should I spend to promote each show?

Even though we have chosen a very simple model, some delay is caused between turns while BASIC calculates the ratings. A significant part of this delay time is caused by the routine (Listing 7-1) that sorts the shows in the order of their ratings for the week. A trade-off was made here in accepting this delay because we felt that the concept of a ratings list was central to the theme of the game and, therefore, must be retained. This sorting routine is an excellent example of where a machine language routine is beneficial in speeding up program execution, but where a BASIC programmer can still achieve satisfactory results. If you are starting to experiment with machine code, it would be a good exercise for you to rewrite this routine in machine

code and improve the performance of Prime Time.

DIRECT CONFLICT SIMULATIONS: WAR GAMES

Another type of game simulates direct combative conflict between opposing forces, usually controlled by either two players or one player and the computer. Games of this sort normally fall into the war game category. Central to the design of this type of game is the resolution of combat between opposing forces. This resolution is normally done through the combat results table (CRT). To illustrate the major part of the design procedure for this type of game, we will design a simple CRT for a fictitious war game and write a BASIC subroutine to calculate its result.

Let us assume we want to simulate a direct conflict between two infantry divisions. We want the result of the battle to be affected by both the relative strength of the forces and the terrain that each force occupies. The first things we must decide are the types of terrain involved, what effects they will have on the battle, and the possible results of the combat.

1. Terrain Types.
Clear
Rough
River

Listing 7-1. Prime Time sort routine.

```
2055 SP=0:REM SORT SHOWS BY POWER
2060 FOR J=1 TO 6
2065 IF PW(BD(NP,J))>=PW(BD(NP,J+1)) THEN 2075
2070 T=BD(NP,J):BD(NP,J)=BD(NP,J+1):BD(NP,J+1)=T:SP=1
2075 NEXT J
2080 IF SP=1 THEN 2055
2100 REM SET TB=BD
2110 FOR J=1 TO 7
2120 FOR I=1 TO 3
2130 TB(I,J)=BD(I,J)
2140 NEXT I:NEXT J
2200 REM GO THROUGH LINEUP
2210 REM CALCULATE BASE VALUE
```

2. Effect of Terrain.

	Attacker	Defender
Clear	=	=
Rough	=	+
River	+	+

- = Has no effect
- + Enhances effectiveness
- Reduces effectiveness

3. Results.

Attacker Eliminated = AE
 Attacker Retaliates = AR
 No Effect = NE
 Defender Retaliates = DR
 Defender Eliminated = DE

Next we place these results in a matrix, or CRT.

Random #	$R < LR$	$R \leq HR$	$R \geq HR$
1	NE	DR	DE
2	AR	NE	DR
3	AE	AR	NE

where:

R = Ratio of attacking force to defending force.

LR = Lower-level force ratio.

HR = Higher-level force ratio.

Note: You assume some value for these parameters, such as LR = .5 and HR = 2. Then during the play-testing phase, these parameters are adjusted to obtain play balance.

Terrain Effect

ATTACKER TOTAL - DEFENDER = NET

For positive values move right one column.

For negative values move left one column.

Now, since we want to incorporate this into a BASIC subroutine, we devise a flow plan for the subroutine.

Flow Plan of CRT Subroutine

Code Values Used:

Terrain.

Clear = 1

Rough = 2

River = 3

Attacker Defender Index.

Attacker = 1

Defender = 2

Results.

1 No Effect

2 Attacker Eliminated

3 Attacker Retreats

4 Defender Retreats

5 Defender Eliminated

For terrain effect, set up as a two-dimensional array [TE(x,y)] during program initialization where x = attacker or defender index and y = terrain type. Also, during program initialization, set up a combat result matrix containing the result code number, using the variable name CR. Make LR = .5 and HR = 2.

Input values to the subroutine are:

AT = Attacker Terrain Number

DT = Defender Terrain Number

R = Ratio of Attacking Force to Defending Force

The subroutine performs the following calculations:

1. Determine CRT row for x,y from random number.
2. Assume column of x=2.
3. If $R < LR$, then x=1.
4. If $R > HR$, then x=3.
5. Get terrain effect from TE and add it to x.
6. Limit x to between 1 and 3.
7. Get result from CR.

Output

RE=Result #

The CRT (Listing 7-2) example program has

```

100 REM INITIALIZATION
110 REM READ IN RESULT #'S
120 FOR I=1 TO 5
130 READ RS$(I)
140 NEXT I
150 REM READ IN TERRAIN EFFECTS MATRIX
160 FOR X=1 TO 2
170 FOR Y=1 TO 3
180 READ TE(X,Y)
190 NEXT Y
200 NEXT X
210 REM READ IN CRT
220 FOR R=1 TO 3
230 FOR C=1 TO 3
240 READ CR(R,C)
250 NEXT C
260 NEXT R
270 REM SET VARIABLE PARAMETER VALUES
280 LR=.5:HR=2
500 REM GET TEST INPUTS
510 INPUT"ENTER AT TER# (1-3)";AT
520 INPUT"ENTER DF TER# (1-3)";DT
530 INPUT "ENTER RATIO (AT PWR/DF PWR)";R
540 GOSUB 1000:REM GET RESULT # FROM CRT SUBROUTINE
550 REM PRINT RESULT
560 PRINT:PRINT RS$(RE)
570 PRINT:PRINT"HIT ANY KEY"
580 GET I$:IF I$="" THEN 580
590 GOTO 500
1000 REM CRT SUBROUTINE
1010 REM GET RANDOM #
1020 R=INT(RND(0)*3+1)
1030 REM ASSUME C=2
1040 C=2
1050 REM CHECK R AGAINST LIMITS
1060 IF R<LR THEN C=1
1070 IF R>HR THEN C=3
1080 REM ADJUST 3 FOR TERRAIN EFFECTS
1090 C=C+TE(1,AT)+TE(2,DT)
1100 REM MAKE SURE C IS 1-3
1110 IF C<1 THEN C=1
1120 IF C>3 THEN C=3
1130 REM GET RESULT #FROM CRT
1140 RE=CR(R,C):RETURN

```

```

30000 REM RESULTS
30005 DATA STANDOFF,ATTACKER ELIMINATED,ATTACKER RETREATS
30006 DATA DEFENDER RETREATS,DEFENDER ELIMINATED
30010 REM TERRAIN EFFECTS TABLE
30011 DATA 0,0,-1,0,1,1
30020 REM CRT
30021 DATA 1,4,5,3,1,4,2,3,1

```

been developed from this program plan. Read through it and see how a simple program plan converts to BASIC coding. For those of you who purchased the loader disk with this book, this BASIC program is contained on the disk file named CRT.

STATISTICAL SIMULATION

The last type of simulation we discuss is the simulation of real-life events through a statistical analysis of data compiled on the subject. A good example of this type of simulation is a sports game. To acquaint you with the design considerations for

this type of game, we describe the design of our game, Baseball's Best.

Baseball's Best, a statistical simulation of the game of professional baseball, uses actual individual player statistics. These statistics are stored in the program data-base and are used to determine the results of all outcomes during the baseball game. An example of the type of statistics used in the game is given in Fig. 7-1.

Name. The player's *name* is given.

POS. Primary and secondary defense *positions* that the player can play are listed.

NAME	POS	SR	D	BA/BCP	TA/PO
GEHRIG	1B,1B	4	2	340	1229
LAJOIE	2B,1B	12	4	338	773
APARICIO	SS,SS	16	5	262	553
BRETT	3B,3B	9	3	306	751
T WILLIAMS	OF,OF	2	2	343	1360
RUTH	OF,OF	4	3	341	1400
DIMAGGIO	CF,OF	2	4	323	1011
JOHNSON	PT,PT	2	1	683	738
GROVE	PT,PT	2	1	594	702
FORD	PT,PT	2	1	626	713
FELLER	PT,PT	2	1	575	696
YOUNG	PT,PT	2	1	637	727
COCHRANE	CA,CA	4	5	319	934
DICKEY	CA,CA	3	5	312	851
FOXX	1B,3B	4	1	325	1154
BOUDREAU	SS,3B	3	4	294	762
COBB	OF,CF	18	2	366	965
MANTLE	CF,1B	5	3	297	1083
B ROBINSON	3B,3B	2	5	267	646
SPEAKER	CF,OF	10	5	344	955

Fig. 7-1. Baseball statistics as used in Baseball's Best.

SR. The *speed rating* is a function of a player's stolen bases and is used in the determination of a player's base-running and stealing capabilities. This number ranges from two to 320, with the higher numbers representing greater speed.

D. A player's *defensive rating* normally ranges from one to five, with one representing below-average fielding, and five signifying one of the all-time best. If you play someone out of position, a defensive rating of zero is assigned. Players with high defensive ratings occasionally make a great play and turn a hit into an out.

BA. The lifetime major league *batting average* is displayed for nonpitchers.

BCP. The *basic control probability* for pitchers is a function of the pitcher's Earned Run Average, the Pitcher's Tired Rating, and whether the pitcher is facing a batter who is hitting from the same side of home plate as the pitcher's throwing arm (i.e., lefty to righty or lefty to lefty, etc.). BCP is used to determine whether the outcome of a pitch is controlled by the pitcher's statistics or by the batter's. BCP begins to be reduced when the PR (Pitcher's Tired Rating) reaches a value of five. Lefty to lefty and righty to righty increases the BCP, whereas lefty to righty or righty to lefty decreases BCP.

TA. *Total average* is the statistical ratio of total bases gained to total outs made. A single and a stolen base are each counted as one base; a home run is counted as four bases. An out counts as one out; a double play as two outs.

PO. *Probability of outs* for pitchers is the measure of the probability that a pitcher will get a hitter to make an out. Hitters against a pitcher with a PO of 740 have a collective batting average of 260.

Baseball's Best uses these statistics in the following manner. After the defensive and offensive options have been selected, the program determines if a special result has occurred.

Special Results

The following special results are possible:

Batter Hit—based on the league statistics for hitting a batter. The batter is awarded first base.

Wild Pitch—based on the league statistics for wild pitches. All runners advance one base.

Balk—based on league statistics for balks. All runners advance one base.

Error—based on the error probabilities by position and on the player's defensive rating. Error probabilities are normally distributed, with a defensive rating of two as the mean.

Batting Results

Whether or not a player gets a hit is based on a series of calculations. The first calculation the computer must make is whether to use the batter's statistics or the pitcher's statistics to determine the results of a play. This calculation considers the BCP (Basic Control Probability), the PR (Pitcher's Tired Rating), the pitcher's throwing arm, and the batter's side of the plate. The PR ranges from zero up and is based on the ratio of the number of batters he faces during this game to the average number of batters the pitcher has faced per game in his career. When the PR equals five, the pitcher's BCP (Basic Control Probability) begins to decline, and at 10 the pitcher has reached his average number of batters faced per game. A pitcher's statistics are most likely to be used to determine the outcome of a play when a pitcher has a high BCP, when he is not tired, and when he is facing a batter who hits from the same side of the plate as the throwing arm of the pitcher.

Next, from either the batter's statistics (Batting Average) or from the pitcher's statistics (Probability of Outs), as determined above, the computer determines whether a hit, out, walk, or strike out has occurred. If a hit-result occurs, the batter's statistics (Total Average) determine if the hit is a single, double, triple, or home run. At this point, the computer also checks the defensive rating of the fielder to whom the ball is hit. There is a small probability that a hit can be turned into an out by a great play or that an out can be turned into a hit by a poor fielding play.

The direction of the flight of the ball is determined by whether the batter is right or left handed. Power hitters (determined by Total Average) hit more long flies. Double plays are based on the

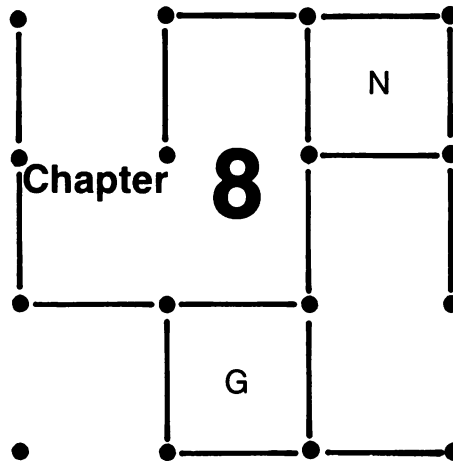
number of double plays into which players of a given speed rating hit. There are no designated hitters in this simulation.

CONCLUSION

This brief look at Baseball's Best's design should give you some idea of the complexity of such a simulation. If you are willing to devote considerable time and effort to such a project and are very knowledgeable about the subject, you should be able to tackle this type of game, even if you are somewhat of a game-design novice. One problem you encounter in designing this type of game is obtaining accurate statistical values for enough

parameters to reasonably represent the subject. It is also difficult to properly balance the many facets of this type of game to achieve a realistic simulation of the subject.

As you can see, designing the simulation models for a microcomputer game is not a simple task. We hope that we have given you enough information to give you a good start in learning this process. Remember that designing a simulation model cannot be done in a vacuum, as the model must work in concert with the artificial-intelligence routines and must be accomplished within the limits of our programming considerations. In the next chapter we explain step 8, developing computer artificial-intelligence models, so read on.



Artificial Intelligence: The Computer Plays

What is artificial intelligence? The term is used in computer parlance to describe a wide variety of ways in which a computer can exhibit human-like reasoning powers. Robotics, medical diagnostics, and computer gaming are just three of the fields that require a computer to exhibit human-like intelligence. In this book, when we speak of artificial-intelligence routines we are referring to the programming of algorithms necessary for the computer to assume the role of a player in a multiplayer computer game.

The development of the computer artificial-intelligence algorithms is the facet of computer game design and programming that we find the most satisfying. These algorithms, which make your computer come alive as an ever-available playing partner, are often quite difficult to define and perfect. However, refining the computer's playing strategy and watching it continually grow in playing strength are most enjoyable experiences.

For the purpose of discussing artificial intelligence, games computers play can be divided into three categories. First, there are games in which

computers do not really contain true intelligence routines but present the player with a computer-moderated challenge or puzzle. Arcade and adventure games fall into this category. Next are games that have fixed objectives, do not require long-term strategic planning, and where individual player moves do not strongly affect the play of other contestants. Word Square is an excellent example of this type of game. Last are games that require true decision making on the part of the computer. Long-term planning is important, and the play of the other competitors must be considered in the decision making. Most abstract strategy games and war games fall into this category. As you might have guessed, these categories are presented in the order of difficulty you face in designing the artificial-intelligence algorithms. Now let us take a closer look at these categories and examine the artificial-intelligence algorithms from some actual games.

PSEUDOINTELLIGENCE

Arcade and adventure games have very limited intelligence routines, if any. This is fairly clear for

games of the arcade variety, but it is not as evident for adventure games that give players the impression that they are conversing with a human-like intelligence. Adventure games in their simplest form have the player converse with the computer in two-word noun-verb sentences. By communicating with the computer, the player travels through a created world to achieve the objectives of the adventure. The computer is not really responding in an intelligent fashion in these games; the responses are programmed into the game by the designer. These games are, in fact, extremely complex puzzles that challenge the player to deduce the proper noun-verb combinations logically in a particular situation so the player can progress toward the ultimate goal. Most of these games are extremely enjoyable to play, but are not really computer-intelligent.

FIXED-OBJECTIVE NONINTERACTIVE SIMULATIONS

Game simulations that have fixed objectives and optimal move determination that is not strongly affected by moves made by the competing players require the simplest artificial-intelligence algorithms. Word Square is a game of this type. The computer's fixed objective is to use the seven random letters together with the letters along the diagonal of the square to achieve the highest possible score. The scoring is determined solely by the computer's performance and is not affected by the words formed by the human players.

Let us outline the objective, the constraints (rules of the game), and the intelligence the computer must display, all of which must be considered in designing the artificial-intelligence routines for Word Square.

Objective: Highest Score

Constraints:

- 1) Words formed must be acceptable Scrabble words.
- 2) Words must be formed with only the random letters together with diagonal letters.
- 3) Each random letter can be used only once.
- 4) All words placed in the square must contain the diagonal letter, and this letter must be retained in its position.

Computer intelligence must be able to find answers to the following questions:

- 1) What makes a valid Scrabble word?
- 2) Is this word legal in the square?
- 3) Is this the best word?

Now let us examine how these considerations are implemented in the Word Square program. There are four BASIC subroutines that implement the artificial-intelligence routines for Word Square. The subroutine at line 1000, Listing 8-1, is called by the main program control loop after the letters for the current turn are displayed on the screen. The first few lines of this routine POKE and ASCII value of these letters into a RAM area, accessible by a machine language routine that fetches the next valid word from the directory. We then set up a FOR-NEXT loop with the index BW running from one to five for the five words of the square. Location 49152 is POKEd with the current word number (BW), and the BASIC subroutine (Listing 8-2) at line 10 is called. This subroutine was originally coded entirely in BASIC, but the execution time for the routine was unacceptable. Therefore we were forced to do some machine language coding. If you plan to write a game in BASIC, you may have to resort to programming a few routines in machine code to achieve satisfactory results. For this reason we reiterate our recommendation that you learn enough machine language to be able to write a program of complexity similar to Listing 8-3.

The first thing the subroutine at line 10 does is call another subroutine at line 800 (Listing 8-4). This subroutine resets a pointer to the beginning of the Wiz's vocabulary list. Then a SYS 49184 in line 20 is executed to jump to the machine language subroutine. This subroutine, Listing 8-3, was previously POKEd into RAM during the initialization phase of the program.

The function of this subroutine is to retrieve the next legal word from the Wiz's vocabulary using the previously outlined constraints that establish legality. Since we have entered only legal Scrabble words into the vocabulary list, we automatically fulfill constraint 1. At the beginning of the source code for this routine is the storage area where BASIC

Listing 8-1. Word Square computer decision subroutine.

```

1000 REM COMPUTER DECISION
1010 FORI=1 TO5: CW$(I)="      ":NEXT I
1014 REM PASS CHRACTERS FOR MACHINE CODE
1015 FOR I=1 TO 5:POKE 49159+I,ASC(MID$(DI$,I,1)):NEXT I
1016 FOR I=1 TO 7:POKE 49167+I,ASC(MID$(LT$,I,1)):NEXT I

1020 SR=0:NW=0:FOR BW=1 TO 5
1030 POKE 49152,BW
1070 GOSUB 10:REM GOTO MACHINE CODE FOR WORD GET
1080 IF MV=0 THEN 1105
1090 NW=NW+1:CW$(BW)=MW$
1100 SR=SR+MV
1105 REM
1110 NEXT BW
1120 TS(0)=SR*NW
1130 RETURN

```

Listing 8-2. Word Square BASIC get-a-word.

```

10 REM COMPUTERS GET  A WORD
15 GOSUB 800: MV=0: MW$=""
20 SYS49184: REM GOTO GETWORD
30 REM GET WORD FROM RAM
40 WD$="": WL=PEEK(49153): FORI=49154 TO 49153+WL: WD$=WD$+CHR$(PEEK(I)): NEXT
50 IF WD$="ZZ" THEN RETURN
60 GOSUB 500: GOSUB 90: GOTO 20

```

Listing 8-3. Word Square machine language get-a-word.

```

1000 ; SOURCE CODE FOR GET WORD ROUTINE
1010 ; PTR POINTS TO CURRENT WORD IN VOCABULARY LIST
1020 PTR=251
1030 *=$C000
1040 ; BW IS WORD # 1-5
1050 BW .BYTE 0
1060 ; WL IS LENGTH OF WORD
1070 WL .BYTE 0
1080 ; WRD IS WORD FROM LIST
1090 WRD .WORD 0,0,0
1100 ; DIA ARE DIAGONAL LETTERS FROM SQUARE
1110 DIA .WORD 0,0,0,0

```

```

1120 ; LTR ARE LETTERS THAT CASAN BE USED TO MAKE WORD
1130 LTR .WORD 0,0,0,0
1140 ; FILE IS A TEMPORARY STORAGE PLACE FOR LETTERS FROM
LTR
1150 FILE .WORD 0,0,0,0
1160 ; CODE STARTS HERE *****
1170 ;
1180 GETWRD
1190 LDY #0
1200 ; FIRST GET A WORD FROM VOCABULARY
1210 ; AND STORE IT IN WRD
1220 LOOP
1230 LDA (PTR),Y
1240 BMI LAST
1250 STA WRD,Y
1260 INY
1270 BPL LOOP
1280 LAST
1290 AND #127
1300 STA WRD,Y
1310 INY
1320 TYA
1330 ; Y= WORD LENGTH
1340 ; STORE LENGTH IN WL AND IT TO VOCAB PONITER
1350 STA WL
1360 CLC
1370 ADC PTR
1380 STA PTR
1390 LDA PTR+1
1400 ADC #0
1410 STA PTR+1
1420 ; IF WORD = ZZ THEN RETURN
1430 LDA WRD
1440 CMP #'Z
1450 BNE CONT
1460 LDA WRD+1
1470 CMP #'Z
1480 BNE CONT
1490 ; RETURN TO BASIC ZZ FOUND
1500 RTS
1510 CONT
1520 ; IF WL<BW THEN GET ANOTHER WORD
1530 LDA WL
1540 CMP BW
1550 BMI GETWRD
1560 ; IF WRD(BW)<>DIA(BW) THEN GET ANOTHER WORD

```

```
1570 LDY BW
1580 DEY
1590 LDA WRD,Y
1600 CMP DIA,Y
1610 BNE GETWRD
1620 SEC
1630 NEXTW
1640 BCC GETWRD
1650 ; IS WRD POSSIBLE ?
1660 LDY #0
1670 LOOP2
1680 LDA LTR,Y
1690 STA FILE,Y
1700 INY
1710 CPY #7
1720 BNE LOOP2
1730 ; LOOP THROUGH WORD
1740 LDY #0
1750 LOOP3
1760 INY
1770 CPY BW
1780 BNE B1
1790 CPY WL
1800 BNE B2
1810 RTS
1820 B1 DEY
1830 B2
1840 LDA WRD,Y
1850 ; GO THROUGH FILE
1860 LDX #0
1870 LOOP4
1880 CMP FILE,X
1890 BEQ LFOUND
1900 INX
1910 CPX #7
1920 BNE LOOP4
1930 ; WORD CAN NOT BE MADE SO GET ANOTHER
1940 CLC
1950 BCC NEXTW
1960 ; LETTER FOUND
1970 LFOUND
1980 LDA #0
1990 STA FILE,X
2000 INY
2010 CPY WL
2020 BNE LOOP3
```

```
2030 ; WORD IS VALID SO RETURN
2040 RTS
```

POKEd the number (BW), the seven random letters (LTR), and the five diagonal letters (DIA) of the square. Upon returning to BASIC from this routine, the next legal word is stored in WRD and its length is stored in WL.

The GETWRD label corresponds to the address 49184 and is the point of entry to the machine code from BASIC. The first thing the machine code does is move a word from the Wiz's vocabulary to WRD. The first time this routine is called for a particular BW, PTR points to the beginning of the Wiz's vocabulary list as set by the subroutine at line 800. The end of a word is signified by the setting of the high bit (>127) of the last character of the word. This method of demarcation is installed by the program that writes out the Wiz's vocabulary file. This method saves considerably more RAM space than a method that uses a separate character, such as a RETURN or a comma, to delimit each word. After the next word is moved to WRD, its length is stored in WL and added to PTR so the pointer points to the next word in the list. The word currently stored at WRD must now be checked against the constraints specified previously to determine if the word is legal within the rules of Word Square.

First, if WRD = ZZ, which marks the end of the vocabulary list, we return to BASIC via the RTS instruction. Next, we compare the length of the word at WRD (WL) with the word number (BW). We know from constraint 4 that WL must be greater than or equal to BW for a word to be legal. Therefore, if WL is less than BW, we branch back to GETWRD and retrieve the next word from the vocabulary list. Also from constraint 4 we know that the character in WRD at the diagonal position must equal the diagonal character. We directly compare

the ASCII values of these characters stored at DIA and WRD, and if they are not equal, we branch back to retrieve the next word.

With the easy checks completed, we must now determine whether or not the word can be formed with the random letters. First, we move the seven random letters to a temporary storage location called PILE. We then compare the characters in WRD one by one against the PILE. We skip over the character in WRD corresponding to the diagonal position since we know from a previous check that it is correct. If the character from WRD is found in the PILE, the character is removed from the PILE, which fulfills constraint 3 that each random letter be used only once. If we cannot find a character in the PILE, this signifies that WRD cannot be formed from the seven random letters.

Failing constraint 2, we branch back to GETWRD through an intermediate branch location (NEXTW) to get the next word. If the last character of WRD is found in the PILE or if it is on the diagonal, then we have met all the constraints. Therefore, we return to BASIC via a RTS with a valid word contained in WRD and the word's length in WL.

After control is returned to BASIC, we move the characters from WRD to the BASIC variable WD\$. If WD\$ = ZZ, then we return to subroutine 1000, or we call the subroutine beginning at line 500 (see Listing 8-5) to evaluate WD\$.

This subroutine adds the letter values (LV) of the characters in WD\$ and stores the sum in the variable V. The value of V is compared to the maximum value of any previous legal word stored in MV. If V > MV, then MV is set equal to V, and MW\$, which contains the characters of the maximum-value word, is set equal to WD\$. This simple procedure gives the computer (Wiz) the intelligence necessary to choose the highest-scoring word from the vocabulary list.

Upon returning to subroutine 1000 from subroutine 10, we first check the value of MV. If it equals 0, then the computer could not find any

Listing 8-4. Word Square vocabulary pointer result routine.

```
800 POKE 251,0:POKE 252,64
810 RETURN
```

```

500 REM EVALUATE WORD
510 V=0:FOR I=1 TO WL
520 V=V+LV(ASC(MID$(WD$,I,1))-64)
530 NEXT I
540 IF V>MV THEN MV=V:MW$=WD$
550 RETURN

```

legal words. Therefore, we jump to the next BW. If MV does not equal zero, then the word corresponding to MV is saved and its score is added to the Wiz's total. This process continues through the five words of the square before the main program control loop resumes. Now you can understand why the Wiz is so smart!

REAL DECISION MAKING

You'll have the most difficulty designing artificial-intelligence routines for a game that requires real decision making on the part of the computer. Games that require long-term strategic planning in which the computer must consider the moves and counter-moves of its opponent fall into this category. Chess is probably the best example of this type of game and has been used for years by the computer-programming community to test artificial-intelligence methodology.

The best way to understand the complexity in designing artificial-intelligence routines is to work through a sample design of such a game. Since the explanation of the design of a game like chess would take a book in itself, we illustrate the process with Dots, one of the simplest games we know.

The object of Dots is to possess more completed square boxes than your opponent has on a playing grid of equidistant dots (See Fig. 8-1).

Players take turns drawing lines to connect two adjacent dots. To claim a box, a player must draw the fourth line that completes a box. When completing a box, players place their initials in the box, score one point, and get to draw another line. See Fig. 8-2. The game ends when all lines are drawn, and the player who has formed the most boxes is the winner.

In the section on the game-design process, we recommended that you develop a rough draft of your program as one step in designing the proper playing strategy for the game. It is not until this point (step 8 in the design process as outlined) that you start to design the computer artificial-intelligence algorithms for your game. In Dots, we can skip these initial steps because the rules of Dots are already fixed. Because Dots can be played without a computer, a list of the important strategy elements necessary for successful play can be compiled. Our discussion of Dots concentrates on step 8 of the design process, which deals with designing the artificial-intelligence routines for your game. If you are designing artificial-intelligence routines for an original game design, you should compile a list for Dots similar to the following one before you start the process described.

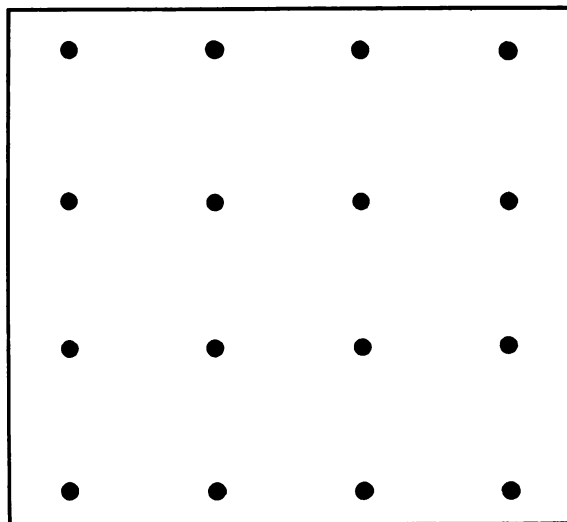


Fig. 8-1. The playing grid for Dots.

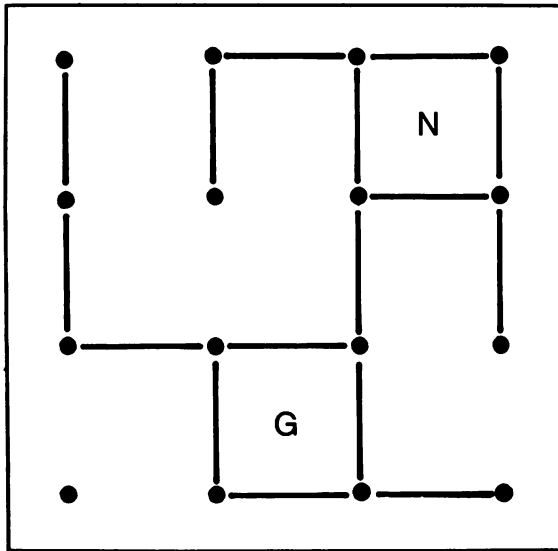


Fig. 8-2. A partially completed Dots game.

Basic Strategy for the Game of Dots

The following list outlines the best strategy for playing dots:

1. If a box can be formed anywhere on the board by drawing a line, then draw that line to form a box.

2. If a box cannot be made, search for a legal move that forms one or two sides of a box.

3. If no lines can be drawn that form only one or two sides of a box, then draw a line that forms three sides.

Advanced Strategy for Playing Dots

The next two points make up advanced strategy for playing Dots:

1. If more than one box can be formed in step 1 of the basic strategy, then choose the box that allows you to form the greatest number of boxes in subsequent moves.

2. If a line must be drawn that forms three sides of a box, then choose to draw the line that allows your opponent to form the fewest boxes in subsequent turns.

Now that we know the strategy, we must teach it to the computer. Listing 8-6 is a program that implements the basic strategy. On the disk loader, the program is contained in the program file DOTS. If you do not have another game to use to practice designing artificial-intelligence routines, it might be fun for you to try to add the advanced strategy for playing Dots to this program.

Listing 8-6. Dots program.

```

50 REM DIM ARRAYS
60 DIM BD(49),BC(49),BX(9,4),BN(24,2),BL(9)
100 GOSUB 20000:REM PROGRAM INITIALIZATION
110 PRINT "{CLR}":PRINT:PRINT "DO YOU WISH TO GO FIRST (Y/N)"
120 GET I$:IF I$="" THEN 120
130 IF I$="Y" THEN CN=2:HN=1:GOTO 150
140 CN=1:HN=2
150 REM
190 NP=0
200 REM MAIN GAME LOOP
205 NP=NP+1:IF NP=3 THEN NP=1
210 GOSUB 5000:REM PRINT BOARD
220 REM CHECK IF ALL LINES IN
230 AL=1
240 FOR I=2 TO 48 STEP 2
250 IF BD(I)=0 THEN AL=0

```

```

260 NEXT I
270 IF AL=1 THEN PRINT "GAME IS OVER":END
280 REM GET COMPUTER'S MOVE IF HIS TURN
290 IF NF=CN THEN 1000
300 REM GET HUMAN'S MOVE
310 PRINT:PRINT:PRINT "PRESS LETTER"
320 GETA$:IF A$="" THEN 320
325 LN=ASC(A$)-64
330 IF LN<1 OR LN>24 THEN 310
331 IF BD(2*LN) THEN 310
400 REM MAKE MOVE
410 BD(2*LN)=1
420 REM CHECK IF IT MAKES A BOX
430 GOSUB 2000
440 IF NB=0 THEN 205
450 FOR I=1 TO NB
460 BT(NF)=BT(NF)+1
470 B=BM(I)
480 IF NF=CN THEN C=67:GOTO 500
490 C=72
500 BC(BL(B))=C:BD(BL(B))=1
510 NEXT I
520 REM GO BACK TO START FOR ANOTHER TRY
530 GOTO 210
1000 REM COMPUTER AI ROUTINE
1010 REM IF MOVE MAKES BOX FG=2 ELSE FG=1
1020 REM LOOP THROUGH ALL LINE LOCATIONS
1030 FG=0:FOR I=1 TO 24
1040 REM CHECK IF MOVE IS LEGAL
1050 IF BD(2*I) THEN 1190
1060 REM DOES IT MAKE A BOX
1070 LN=I:BD(2*I)=1:GOSUB 2000
1075 BD(I*2)=0
1080 IF NB<>0 THEN FG=2:I=24:GOTO 1190
1090 REM DOES IF MAKE THREE IGNORE IF ALREADY FOUND A MOVE
1100 IF (MN=3 AND FG=0) OR (MN<3) THEN FG=1:TL=I
1190 NEXT I
1200 REM IF BOX MADE JUST EXIT
1210 IF FG=2 THEN 400
1220 REM ELSE SET LN=TL
1230 LN=TL
1240 GOTO 400
2000 REM ROUTINE CHECK IF MOVE MAKES A BOX
2010 NB=0:BM(1)=0:BM(2)=0:MN=0
2020 FOR II=1 TO 2
2030 B=BN(LN,II)

```



```

2040 IF B=0 THEN 2500
2050 SM=0
2060 FOR J=1 TO 4
2070 SM=SM+BD(2*BX(B,J))
2080 NEXT J
2090 IF SM=4 THEN NB=NB+1:BM(NB)=B
2100 IF SM>MN THEN MN=SM
2500 NEXT I
2999 RETURN
5000 REM PRINT BOARD
5010 REM POSITON CURSOR
5020 PRINT "{CLR}";:GOSUB 6000
5030 REM PRINT #'S
5040 PRINT "{YELD}";
5050 PRINT TAB(17)" A B C BOX TOTALS"
5060 PRINT TAB(17)"D E F G HUMAN"
5070 PRINT TAB(17)" H I J ";BT(HN)
5080 PRINT TAB(17)"K L M N"
5090 PRINT TAB(17)" O P Q COMPUTER"
5100 PRINT TAB(17)"R S T U ";BT(CN)
5110 PRINT TAB(17)" V W X"
5120 REM REPOSITION CURSOR
5130 GOSUB 6000
5135 PRINT "{WHT}"TAB(17)
5140 REM PRINT OUT BOARD
5150 K=0
5160 FOR I=1 TO 7
5170 FOR J=1 TO 7
5180 K=K+1
5190 IF BD(K)=1 THEN PRINT CHR$(BC(K));:GOTO 5200
5195 PRINT "{C/RT}";
5200 NEXT J
5210 PRINT:PRINT TAB(17)
5220 NEXT I
5999 RETURN
6000 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}";:RETURN
20000 REM PROGRAM INITIALIZATION
20010 REM READ IN DATA FOR ARRAYS
20020 FOR I=1 TO 49
20030 READ BD(I),BC(I)
20040 NEXT I
20050 FOR I=1 TO 24
20060 FOR J=1 TO 2
20070 READ BN(I,J)
20080 NEXT J

```

```

20090 NEXT I
20100 FOR I=1 TO 9
20110 FOR J=1 TO 4
21120 READ BX(I,J)
21130 NEXT J
21140 NEXT I
21150 FOR I=1 TO 9
21160 READ BL(I)
21170 NEXT I
22000 BT(1)=0:BT(2)=0
29999 RETURN
30000 REM DATA FOR BD,BC ARRAYS
30010 DATA 1,113,0,99,1,113,0,99,1,113,0,99,1,113
30015 DATA 0,98,0,0,0,98,0,0,0,98,0,0,0,98
30020 DATA 1,113,0,99,1,113,0,99,1,113,0,99,1,113
30025 DATA 0,98,0,0,0,98,0,0,0,98,0,0,0,98
30030 DATA 1,113,0,99,1,113,0,99,1,113,0,99,1,113
30035 DATA 0,98,0,0,0,98,0,0,0,98,0,0,0,98
30040 DATA 1,113,0,99,1,113,0,99,1,113,0,99,1,113
30100 REM DATA FOR BN ARRAY
30110 DATA 1,0,2,0,3,0,1,0,1,2,2,3,3,0,1,4
30120 DATA 2,5,3,6,4,0,4,5,5,6,6,0,4,7,5,8
30130 DATA 6,9,7,0,7,8,8,9,9,0,7,0,8,0,9,0
30200 REM DATA FOR BX ARRAY
30210 DATA 1,4,5,8
30220 DATA 2,5,6,9
30230 DATA 3,6,7,10
30240 DATA 8,11,12,15
30250 DATA 9,12,13,16
30260 DATA 10,13,14,17
30270 DATA 15,18,19,22
30280 DATA 16,19,20,23
30290 DATA 17,20,21,24
30300 REM DATA FOR BL ARRAY
30310 DATA 9,11,13,23,25,27,37,39,41

```

Program Variables Used in Dots

The following variables are used in Dots:

I\$—A temporary string location used for I/O.

CN—Player number of computer player.

HN—Player number of human player.

NP—Number of current or active player.

AL—Flag = 1 if all lines are drawn.

A\$—Temporary string used for I/O.

LN—Line number (1-24) corresponds to position

A-X on board.

NB—Number of boxes formed when a line is drawn.

B—Temporary variable containing box number.

C—Variable containing ASCII value for H or C, which is to be printed in the box.

MN—Number of sides of a box that exist after a line is drawn on the box.

TL—Temporary line number.

I—Loop counter.
 II—Loop counter.
 J—Loop counter.
 SM—Sum used in counting sides of a box.

Array Variables Used in Dots

The following array variables are used in Dots:

Name Dimension Function

BD	49	Playing board—contains flag that = 1 if spot is occupied.
BC	49	Character that is printed if BD flag = 1.
BX	9,4	Contains the four line numbers that make up each of the nine possible boxes.
BN	24,2	Contains the box number in which each line is contained. If the second entry = 0, then the line is contained in only one box.
BL	9	The board location for each of the nine boxes.

Subroutines Used in Dots

The following subroutines are used in Dots:

Line Number Function

1000	Determines computer's move or line number, on exit LN = computer's move.
2000	Determines if move forms a box. On entry LN = line number of move. On exit NB = number of boxes made.
	MN = maximum number of lines in box.
5000	Prints the playing board.
6000	Positions cursor.
20000	Initializes program variables.

The subroutines that implement the basic

strategy for Dots are 1000 and 2000. When the main program loop determines that it is the computer's turn, the subroutine beginning at line 1000 is called for the computer's move. When control is returned, the main program expects the variable LN to contain a legal line move for the computer. The first thing this subroutine does is set the flag variable FG = 0, which indicates that the computer has not yet found an acceptable move. Next, it starts a FOR-NEXT loop with the I counter running from 1 to 24, which represents the 24 possible line moves. Within the I loop, the subroutine first checks to see if the BD array flag is set for line I. If it is set, then this move has already been made. A jump is then made to the end of the loop. If the BD flag is not set for this location, then the line number I is a legal move. The computer's move LN is set equal to I, and the subroutine at line 2000 is called to determine if the move LB forms a box.

The subroutine at line 2000 begins by setting to zero the number of boxes made (NB), the box-made location number (BM), and the maximum number of sides in the box (MN). Next, for each possible box that LN could form, the subroutine adds the number of sides in that box, using the variable SM to maintain the sum. If SM = 4, a box has been formed, so NB is incremented by one, and BM(NB) is set equal to the box-number location of the box that has been formed. Finally, inside the loop SM is compared to MN. If SM is greater, then MN is updated. When control is returned to the subroutine beginning at line 1000, NB equals the number of boxes made and MN equals the number of sides formed by the line-move number LN. Now the program checks the value of NB. If NB does not equal zero, the flag FG is set equal to 2 to indicate that a box has been made. The Loop counter I is set equal to 24 to force an exit from the loop, and a jump is made to the end of the loop.

This sequence of commands implements the first element of the basic strategy, since our primary goal is to make a box if possible. If NB equals zero, a check is made for two possible conditions. The first condition is when MN = 3 and a legal move has not yet been found. The other condition is when MN is less than 3. If either of these condi-

tions exists, then the move-found flag FG is set equal to 1, and the temporary line-move variable TL is set equal to I. The loop is continued until either a box is formed or all 24 possible line moves have been investigated. At the end of the loop, the flag FG is checked. If FG is equal to 2, then a box was found and control is immediately returned to the main program because LN already contains the line-move number that found the box. If FG does not equal 2, then a box was not formed, and the computer line-move number LN is set equal to the last temporary line move TL that was found. Control is then returned to the main program loop. Careful note should be taken of the conditional statement that sets TL.

IF (MN = 3 and FG = 0) or (MN < 3) then
 FG = 1:TL = I

This condition is set so that a third side is added to a box only if no previous legal line move is found (FG = 0). If a previous line move has already been found, then TL is only changed by a line move that forms only one or two sides to a box. This condition assures that the third side of a box is never drawn unless no legal lines can be drawn to form only one or two sides. This implements elements 2 and 3 of the basic strategy. We can implement elements 2 and 3 with a single program loop by always accepting a line move that forms a two-sided box, and by only accepting a line move that forms a three-sided box if no legal move has yet been found. If we perform 2 and 3 sequentially, we have to use another FOR-NEXT loop, which adds considerably to the execution time. The single loop approach can not be taken to implement the advanced strategy because element 2 of the advanced strategy requires an evaluation of all possible three-sided box moves to choose the best move.

Dots is a somewhat simplistic game, but it is a good example of how a computer can display intelligence. After you go through this example carefully, we think you will begin to see the complexity involved in developing artificial-intelligence algorithms. These algorithms become increasingly difficult when the computer has to consider subse-

quent moves by its opponent. This becomes evident if you attempt to add the advanced strategy to the game of Dots.

As you can tell from the amount of time it takes BASIC to execute the basic strategy for a small grid, a large grid version of the Dots program implementing the advanced strategy would have to be programmed in a higher-efficiency language to be practical. For this reason, computer artificial-intelligence routines often use a *weighting-factor* evaluation to determine the desirability of a particular move since weighting-factor evaluations are much simpler and execute much faster. To understand the weighting-factor approach, let us go through a simple example.

Weighting-Factor Evaluations

Assume we want to design an artificial-intelligence algorithm for a computer football simulation. In our simple simulation, we want to teach the computer to select the proper defensive formation from one of two choices: PASS defense or RUN defense. The first thing we do is list the factors the computer uses to select the defense.

 DOWNS LEFT (D) = (5 - DOWN#)
 YARDS TO GO FOR FIRST DOWN (Y)

Recognizing that D and Y are not mutually exclusive parameters and that relative values are important, we write the following evaluation equation

$$E = (W1 \cdot D) + (W2 \cdot Y/D)$$

where E is the evaluation score, W1 is the weighting factor applied to number of downs left, and W2 is the weighting factor applied to ratio of yards to go/downs left.

Now we arbitrarily choose a positive value of E to indicate a pass defense and a negative value of E to indicate a run defense. Using this convention, we estimate some values for W1 and W2 by choosing two situations of varying down and yard-to-go situations: where it is slightly more probable that the offense will call a pass rather than a run (E = 1), and where it is slightly more probable that the offense will call a run rather than a pass (E = -1).

DOWN	DOWNS LEFT	YARDS TO GO	E =
2	3	6	-1
3	2	3	1

This gives us a set of two simultaneous equations with two unknowns, W1 and W2.

$$\begin{aligned}-1 &= W1 * 3 + W2 * 6/3 \\ 1 &= W1 * 2 + W2 * 3/2\end{aligned}$$

Solving these equations, we find that W1 = -7, and W2 = +10, so our equation for determining pass or run defense is

$$E = 10Y/D - 7D.$$

You can then program this equation into your football game for the computer to determine whether to call a pass or run defense. This is just a simplified example that shows how to develop a weighting-factor formula. If you were actually developing such an algorithm for a football game, you would have to add additional parameters and a random factor into the computer's decision making so that its play calling would not be absolutely predictable by the human player. You would then adjust these weighting factors during the play-testing phase to give the proper play-calling balance. It is obvious from the following examples that even such a simple factor-weighting model can produce satisfactory results.

DOWN	D	YARDS TO GO	Y/D	E =
2	3	10	10/3	-12.3
3	2	5	5/2	11
2	3	5	5/3	-4.3
3	2	1	1/2	-9

DEFENSE CALLED
PASS
PASS
RUN
RUN

By comparing the weighting-factor model used for our football simulation with the algorithm required to implement the advanced strategy for the

Dots game, you can see that the weighting-factor method is much simpler than a method that must consider counter-moves by the opponent. If the game you are designing does not require the computer to make radically different decisions that depend on the possible replies by its opponent, then we recommend that you first try to use a simple weighting-factor model for the computer artificial-intelligence routines. Only if this approach fails should you attempt to follow forward-search and move-evaluation techniques.

ADVANCED TECHNIQUES IN ARTIFICIAL INTELLIGENCE FOR TWO-PLAYER GAMES

The artificial-intelligence techniques we have described are quite good for multiplayer games and games that are not highly dependent on future plays by an opponent. These techniques can be implemented quite simply in BASIC. However, there are certain games, such as chess, checkers, and backgammon, that cannot be modeled very well with these techniques because the selection of the optimal move in these games is dependent on your opponent's response to your move, on your possible responses to that move, and so on. To develop a computer program capable of competing with strong human players in these types of game, you must employ more powerful algorithms that are capable of evaluating the future consequences of a move.

This section introduces a technique you can use to program such a game. We use chess as the example in our discussion of evaluation functions, tree generation, and tree searching, which are necessary parts of this technique. If you want to embark on a major endeavor, you can read some of the considerable literature available about programming a computer to play chess. BASIC programming is much too slow to implement the algorithms we describe in this section. At the very least, you need to use a fast compiled language such as Fortran or Pascal, and you must use machine language to develop very strong programs.

Evaluation Function

An evaluation function is an equation,

algorithm, or subroutine in a computer program that can assign a numerical value to any possible position that can occur in the game. It is a necessary element in choosing a good move because we need to be able to associate a score with each move based on the positions that are possible subsequent to the move. Once we have a score for each of our possible moves, we can simply choose the move with the highest score (we assume our evaluation function is such that the higher the score, the better the position is for the player deciding on the move). The weighting-factor algorithm we described previously is often used to implement the evaluation function. Let us now look at the simplest example of such a function for the game of chess.

Our chess evaluation function contains a material score (MA), a mobility score (MO), and a weighting function (W), which are combined to determine a total score (TS). The algebraic equation representing this function is as follows:

$$TS = MA + W * MO$$

Before we can evaluate the total score, we need to calculate values for MA, W, and MO. To calculate MA, we write a subroutine that numerically sums up the value of the white and black pieces based on traditional piece evaluations: queen = 9, rook = 5, bishop = 3, knight = 3, pawn = 1. The king should be given an extremely high number; its value is actually ∞ because losing your king means losing the game. If we assume the computer is playing the white pieces and that a higher number indicates a better position, then we would calculate the material score as follows:

$$MA = \text{white piece total} - \text{black piece total}$$

To calculate the mobility score, we assume it is directly related to the number of legal white and black moves in a given position and calculate as follows:

$$MO = \text{number of white moves} - \text{number of black moves}$$

The determination of the weighting factors in

the evaluation function for games such as chess is not straightforward because their optimal values are not amenable to precise calculation. Hence, they are usually determined by a trial and error approach. To evaluate W for our simple example, we could develop two different evaluation functions, one using a large value of W and the other a small value of W. For example:

- 1) $TS = MA + .9 * MO$
- 2) $TS = MA + .1 * MO$

We would then play a number of games against the program, using each evaluation function, or we could pit the different versions of the program against one another. We then adjust the values of W closer to the W in the better model, and we would continue this process until the difference in the performance of the program within a range of values of W is not perceptible. We now set W to a value within this range. This method produces a value of W very close to the optimum, given the limitations of the evaluation function itself.

Growing Trees

The evaluation function is a static evaluator that determines a score for a given game position. To determine a score for a given move, we must evaluate the subsequent positions that can arise after the move we are evaluating. To perform this evaluation, we construct a “tree” of game positions. At the root of the tree is the position arising from the move under consideration. The size of the tree is determined by the number of moves, or ply, that we consider in the future. At the terminal nodes, or branches, of the tree are all the possible positions at the maximum ply depth that can arise from the move we are evaluating. An example of such a tree is illustrated in Fig. 8-3. This illustration assumes that in the current position we have two possible moves, m1 and m2, from which we want to choose the better move.

In this simple example, there are two possible replies at each position, or node, in the tree. In artificial-intelligence jargon, this is usually called the *branching factor*. Hence, our tree has a branching

factor of 2. We examine all moves possible in the current position m1, m2, then all of our opponent's replies to these moves, and finally all of our possible replies to our opponent's moves. Thus, we look three moves into the future, and our tree is said to be searched to a depth of 3 ply. For a tree with a constant branching ratio (b), the number of terminal nodes of the tree equals $b^{(\text{ply depth})}$. In our example, the number of positions at the branches of the tree equals 2^3 , or 8. To determine the best move between m1 and m2, we evaluate the terminal positions to assign a score to m1 and m2. Then we choose the move with the highest score.

Searching the Tree

How do you search this tree to determine the best move? The simplest method is to examine all the terminal nodes of the tree. If a higher score is better for the player about to choose between m1 and m2, then it is the objective of this player to choose the highest-scoring move at odd ply and for

the opponent to choose the lowest-scoring move at even ply. In Fig. 8-3 we indicated the score that our evaluation function would calculate at each position. If we evaluate the position (p111) arising from the move sequence m1, m11, m111, then we calculate a score of 8. To determine the highest achievable scores for m1 and m2, we search the tree as follows:

1. First, we evaluate p111 and p112, all the positions that are possible following our opponent's move m11. We want to maximize the score at this ply, so we choose move m111, whose score is 8, and we assign this score to move m11.
2. Next we follow the same process for move m12 and arrive at a score of 9 for move m12.
3. It is our opponent's objective to minimize the score at even plies, so we assume the opponent would choose move m11 because it has a lower score than move m12. Therefore, we assign the value 8 to move m1.
4. Now we examine the tree emanating from

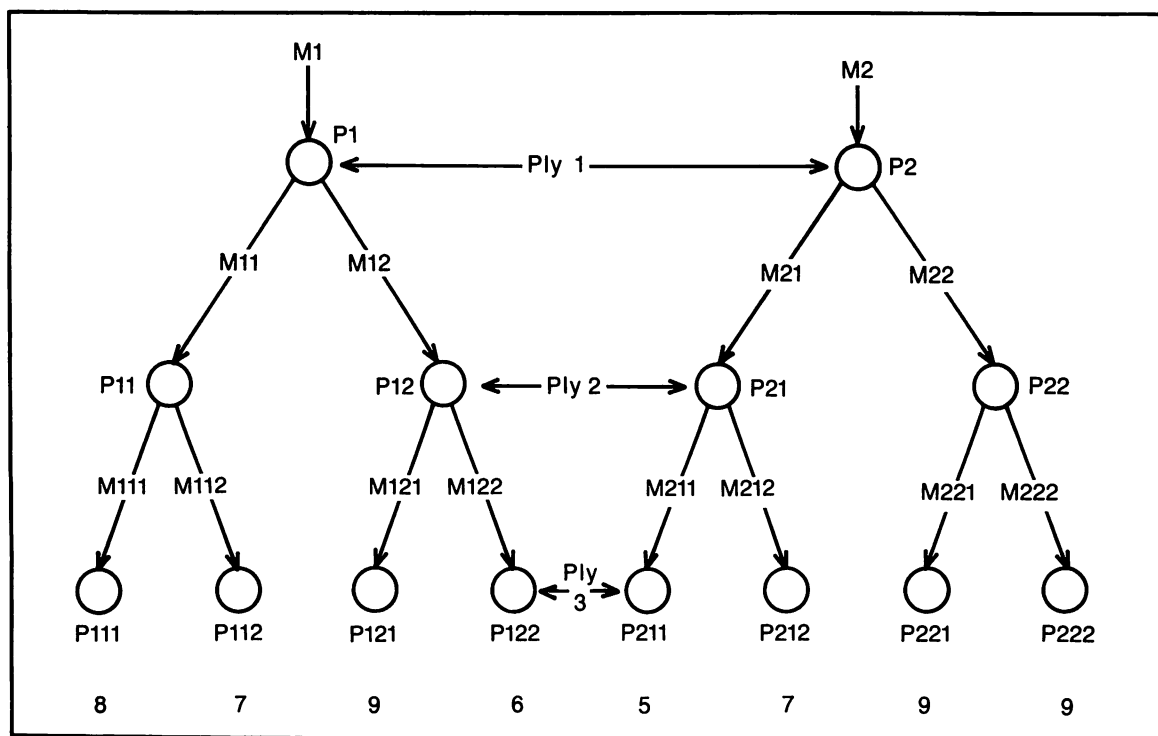


Fig. 8-3. A chess tree.

move m2 in exactly the same way we searched the tree emanating from move m1. In order, m21 equals 7, m22 equals 9, and m2 equals 7.

5. Finally, since m1 (8) is greater than m2 (7) and we want to maximize our score at this ply, we choose the move m1.

This method of searching the tree is simple, straight-forward, and, in practice, has the advantage of requiring very little computer memory. It is quite adequate for games, such as tic-tac-toe, that have small trees. However, this method suffers from the fact that we are required to evaluate every one of the terminal nodes of the tree. For example, if we want to perform a 4-ply search in a game of chess that has a branching ratio > 30 , then the number of terminal positions we have to evaluate equals 30^4 , or 810,000 positions. Even with the simplest of evaluation functions, this would take a prohibitive amount of time on a microcomputer. Fortunately, by assuming that players always make those moves that either maximize their score or minimize their opponent's, we do not have to examine every node in the tree. Now let us examine such a searching process.

1. First we examine all the positions arising from move m11 and choose the highest score of 8.

2. Next we evaluate position p121. Finding it has a value of 9, we terminate our search and do not evaluate any further moves emanating from the position p12, which was reached following m12. The reason for the termination is that, since our opponent wants to minimize the score, move m12 would not be chosen because it would result in a score greater than 8, which was the maximum achievable following move m11.

3. Having completed our search of this portion of the tree, we assign a value of 8 to move m1.

4. Next we evaluate p211, which equals 5, and p212, which equals 7, then choose the maximum and thereby assign a score of 7 to move m21.

5. We can now terminate the rest of the search because we already know that m1 can achieve a score of 8 and because, at best, m2 can only achieve

a score of 7, there is no need to evaluate the remainder of the tree.

Comparing the two methods of search, we can see that the first method requires us to evaluate 8 nodes, while the second method requires us to evaluate only 5 nodes. In practice the second method of searching the tree is implemented by a structured process known as the Alpha-Beta algorithm. It can be shown that for an optimally structured tree, the Alpha-Beta algorithm requires the evaluation of approximately $2 * \sqrt{N}$ positions, where N is the total number of terminal nodes in the tree. For the 4-ply-deep chess search, we need to evaluate approximately 1800 positions instead of 810,000 positions—quite a savings!

An optimally structured tree is one in which the strongest moves for each side are always examined first. We cannot sort the moves in perfect order unless we first evaluate all the positions, which would defeat the purpose of first ordering them. However, many different techniques have been developed to achieve a close-to-optimum ordering of moves without actually evaluating all of them. A discussion of these techniques is beyond the scope of this book. You can refer to *Computer Gamesmanship* by David Levy, Simon and Schuster, 1983, for a simple introduction to these techniques.

Once you master the techniques of setting up the evaluation function, constructing the move tree, and searching the move tree, you are ready to write a program to play games such as checkers, backgammon, and chess. The playing strength of your program will depend primarily on two factors:

1. Your understanding of the game, since better knowledge of the game should allow you to develop a more accurate evaluation function.

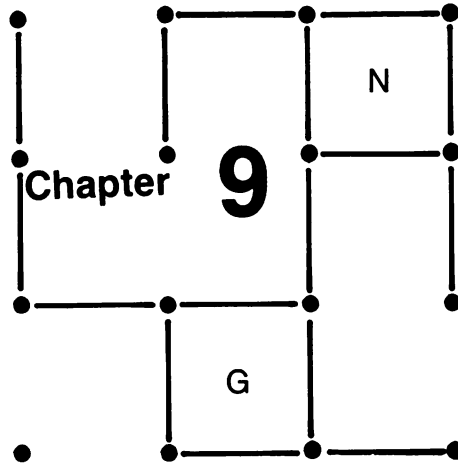
2. Your skill as a programmer, since the more efficient your program is, the more positions you can examine in a given amount of time.

Developing a strong program of this type is one of the most difficult tasks a game programmer is likely to attempt. We do not advise you to undertake such a project unless you are willing to devote

many months or possibly years on developing and refining your program.

We hope you will benefit from this explanation of artificial intelligence. We have tried to restrict our discussion to those types of models that are most readily developed and to those that most likely could be used in a simulation-type strategy game. How-

ever, we have only scratched the surface of this most exciting field. You should be able to find a wealth of additional information on this topic. As in the case of designing a simulation model, artificial-intelligence routines must be developed within our programming limitations. The next chapter is devoted to these programming considerations.



Programming Considerations

As you design a game on a microcomputer, you must be mindful of both the advantages and the limitations of the programming language you are using. You must also plan your program well and make it flexible enough for you to incorporate the numerous changes in your initial design that will surely be required. In this chapter, we describe the limitations imposed on game design by computer RAM and discuss the advantages and disadvantages of three common computer languages, assembly language, FORTH, and BASIC. We describe good programming practices for the design of program I/O, program flexibility, and program structure. Finally, we describe methods for improving program execution speed and for reducing RAM requirements.

LIMITATIONS IMPOSED BY COMPUTER RAM

Just a few years back most microcomputer systems had 16K RAM or less. This RAM size placed severe limitations on the design of computer games. To write a program of more than limited

scope, the programmer had to cope in assembly language and use all sorts of tricks to fit the design into the computer. Today most home computers have at least 48K of RAM, so if you write a game for your own use of the type presented in this book, you do not have to concern yourself with space saving or optimized coding and programming tricks. However, if you progress to the point of developing a commercial product or are planning to write a game that contains hi-resolution graphics, then you need to plan your project carefully, considering the advantages and disadvantages of various programming languages. You may also have to resort to some of the programming tricks discussed at the end of the chapter.

CHOOSING A PROGRAMMING LANGUAGE

Choosing the programming language you will use to code your game is an extremely important decision in the game-design process. You must consider numerous trade-offs between code efficiency and programming development time. In addition,

the language you choose often dictates the best way to model a simulation. We discuss the advantages and disadvantages of using assembly, FORTH, and BASIC programming languages for game design. These languages are readily available for your computer, and they cover the ranges of programming ease and efficiency.

Assembly Language

Assembly language is not really a language in itself; rather it is a term used to refer to the use of a set of mnemonic representations of the native machine code of the microprocessor. To write in machine code, you need an assembler that automatically converts the mnemonic representations to the native machine code. Assemblers also aid in the programming process by allowing you to use labels instead of exact memory addresses and by allowing you to use various directions to control the assembly process. It is beyond the scope of this book to fully explain the process of writing machine language code, so we limit the discussion to the advantages and disadvantages of using machine code.

The principal advantage of machine code is program execution speed. The microprocessor is directly instructed to perform a task, so time is not wasted executing code to consider a host of possibilities before deciding which task to perform. For example, a simple programming loop executes 200 to 1000 times faster in machine code than in BASIC. Properly coded machine language tends to be much more compact than BASIC. Assembly language also produces more compact code than FORTH for smaller projects. For large applications, however, FORTH is probably the most compact of all languages.

The disadvantages of using assembly language is that much greater programming time is required. Following is a list of factors that contribute to the lengthy development time.

- When you program in machine code, you find the machine is very dumb. It does not know how to print, take a square root, or interpret any of the high-level expressions you commonly use in BASIC. Depending on how much you make use of

machine code subroutines packed away in your computer's ROM, you may have to write as much as several hundred lines of code simply to print a message on the screen.

- Since the code is written in such a low level, it is extremely difficult to read and understand. The programmer must supply considerable written comments in the source code so the code can be followed—even by the programmer who wrote the code.

- Every time a change is made in the *source*, or assembly language code, the programmer must reassemble the source to produce the *object*, or machine code to test the program.

- It is extremely difficult to debug a machine language program. There are no error messages to guide you to the problem. It is fairly easy to make an error that causes your machine to lock up, requiring you to power down and start all over.

For these reasons, we recommend that, whenever possible, you avoid writing your game in assembly language. It takes much too long, and you can easily become frustrated before you get the satisfaction of seeing your design come alive on the screen. Assembly language, however, does make an excellent partner to BASIC. If you have the time and inclination, we do recommend that you learn to code small assembly language routines that can be used to help BASIC in those difficult places. A good example of this is the small machine code routine used in Word Square to retrieve a valid word from the computer's dictionary.

FORTH

FORTH is a difficult language to describe since it is not only a language but an operating system and assembler as well. To make matters more difficult, no two FORTHs are exactly the same because FORTH is an extendable language. The fact that it is extendable gives FORTH its real power. The most basic form of FORTH consists of a kernel of the most common operations required in programming. These operations, called *words*, are really subroutines coded in machine language. The collection of FORTH words currently available to the pro-

grammer is known as the *dictionary*. The great thing about FORTH is the capability to expand the dictionary directly in machine code or by defining new words from an existing word. This combination allows you much control in the development stage. You can program an entire application using only high-level words made up of existing FORTH words. Then if program execution speed is unacceptable for a portion of the program, you can recode the time-critical words of the section in machine code.

As in the case of assembly, it is beyond the scope of this book to describe fully the working of FORTH, but again we briefly summarize the advantages and disadvantages that we have been able to delineate. First, in our opinion, FORTH is the best language for programming microcomputer games—if you could learn how to use it! Unfortunately, learning how to use FORTH is not easy. This problem is compounded because there are relatively few FORTH teachers. Because of this, there is a scarcity of good FORTH utility programs like those available to the BASIC programmer. FORTH is a cryptic language that is extremely difficult for most novice programmers to learn.

If you spend the time to learn and use FORTH properly, you'll find the language has three outstanding qualities:

1. By its nature, FORTH forces structured programming, which forces the programmer to develop a logical and smooth-flowing program.
2. FORTH code is very easy to modify late in the design stage. When you change the definition of a single word, all the other words that make use of this word in their definitions adapt to the change. This makes it easy to transfer FORTH applications between different microcomputer systems.
3. You can make FORTH code execute almost as quickly as pure machine code by coding time-critical words directly in machine code.
4. Because a FORTH application is built one word at a time, and because each word can be interactively tested at the terminal, FORTH is extremely easy to debug.
5. Because FORTH code is easy to modify, a programmer can often save a great deal of develop-

ment time by modifying previously developed code for a new application.

FORTH is our favorite language because it gives the programmer great control of the machine and still considerably reduces program development time. We believe it may be well worth your time to become familiar with the fundamental workings of FORTH and decide if it appeals to you. However, keep in mind that we spent many years of hard work to attain a fluency in this language. Therefore, unless you are extremely dedicated or want to produce a commercial product, we do not recommend you use FORTH.

BASIC

BASIC, as you might have guessed by now, is the language we recommend for most, if not all, of your game designs. To give you an idea of the types of BASIC programs that are possible, the application programs in this book are coded almost entirely in BASIC. There seems to be a certain snobbish attitude among some microcomputer programmers that if a program is written in BASIC, it must be slow, inefficient, and of low quality. You should avoid the temptation to use some exotic language to make your game appear "professional." As a matter of fact, if BASIC is supplemented in the proper places with machine code routines, a very professional product is impossible.

The disadvantages of BASIC are fairly well known.

1. It is extremely slow in program execution because it is an interpreted language rather than a compiled language.
2. BASIC source code is often very RAM intensive.
3. Most long programs have many GOTOs and GOSUBs, which give a BASIC program a poor structure. This flow of poorly structured code is often difficult to follow and very hard for another programmer to modify.

The advantages of BASIC are the following:

1. It is extremely easy to use, and many high-level functions are well supported.

2. It is readily available because it comes with your machine, and, therefore, most programmers are familiar with it. There are also many utility programs available to aid the BASIC game designer.

3. Since BASIC programs do not have to be compiled, program changes can be made and tested interactively at your terminal.

4. BASIC errors are well trapped and fairly easy for the programmer to trace.

5. The development of a new BASIC application requires about 1/10 to 1/20 of the time of an equivalent machine code project.

As you have gathered by now, this book does not try to teach the art of computer programming. We assume that you already have a good working knowledge of how to program in BASIC. If you are interested in learning assembly language, we suggest that you read *6502 Machine & Assembly Language Programming*, by Mike Smith, copyright 1984 by TAB BOOKS Inc. If you are interested in learning FORTH, another TAB title, *Beginning FORTH*, by Paul Chirlian is very useful.

The rest of this chapter concentrates on the program-implementation phase of the game-design process. While many recommendations are language independent, we make the assumption that we are discussing BASIC design implementation.

PROGRAM I/O DESIGN

The most important programming consideration in the development of a microcomputer game is the design of the input and output routines. To play the game, a player must constantly interact with these routines, so it is paramount that the input and output of a game be as clear and simple as possible. Fancy graphical input schemes may be impressive the first few times you play a game, but if they add to the complexity of input, then they will detract from the game. The input/output routines should not prevent the user from devoting full attention to the playing of the game.

The two principal input devices currently used in microcomputer games are the keyboard and the joystick. The joystick has two advantages over the keyboard. First, a joystick is a sensory-enforced input device. For games such as war games or arcade games that require directional movement commands, it is much clearer and faster for a player to associate the forward movement of a joystick with an upward movement on the playing screen than to associate such movement with pressing the letter U on the keyboard. The other primary advantage that joystick input has over the keyboard is that joysticks allow much greater control to be maintained by the program. Since many fewer input values are possible from a joystick, it is much easier and faster to process the input, and there are far fewer improper inputs that need to be checked and corrected. If you have never used a joystick in one of your programs, there is a very simple example of joystick use in Listing 9-1, which is contained in the BASIC file JOY on the loader disk.

The limited number of input values possible with a joystick is also one of its main disadvantages for games such as Takeover. This type of game requires a great range of input decisions and numerical input that are not sensorily related to joystick directions. Another small disadvantage of designing a game around joystick input is that not all computer users have joysticks. This really should not affect your decision, however, because you can always design a routine that allows for optional keyboard entry.

If you choose to design a game that is better suited to keyboard entry, you must be careful to give the user clear instructions and to provide easy navigation through the program's control structures. A good method for achieving this is to use a menu control scheme like the one in Takeover. In Takeover you are presented a command menu that lists all the currently available options. Using a single keystroke input, such as S for sell stock, the user is able to perform various transactions as simply as possible. This type of input is used as much as possible except when an INPUT statement is to be used to input actual dollar and share amounts. The INPUT statement should be avoided as much as possi-

Listing 9-1. Example of joystick use.

```
100 REM USING THE C-64 JOY STICK
110 SL=56320:REM LOCATION OF JOY STICK PORT PORT2
150 PRINT "{CLR}"
190 PRINT "{C/LF}";
200 REM GET JOYSTICK VALUE
210 GOSUB 1000
220 REM MOVE CURSOR ACCORDINGLY
230 ON JV+1 GOSUB 500,510,520,500,540,550,560,500,580,590,
600
250 REM GET ANOTHER VALUE
260 GOTO 190
500 RETURN
510 REM UP
511 PRINT "{C/UP}";:RETURN
520 REM DOWN
521 PRINT "{C/DN}";:RETURN
540 REM LEFT
541 PRINT "{C/LF}";:RETURN
550 REM LEFT & UP
551 PRINT "{C/LF}{C/UP}";:RETURN
560 REM LEFT & DOWN
561 PRINT "{C/LF}{C/DN}";:RETURN
580 REM RIGHT
581 PRINT "{C/RT}";:RETURN
590 REM RIGHT & UP
591 PRINT "{C/UP}{C/RT}";:RETURN
600 REM RIGHT & DOWN
601 PRINT "{C/RT}{C/DN}";:RETURN
1000 REM GET STICK VALUE
1010 JV=15-PEEK(SL)AND15
1020 RETURN
```

ble because it transfers the most control from the program to the user.

If you are using a keyboard menu-input structure in your program, you must always check each input value to determine if it is acceptable and legal within the rules of the game. Should a user input an illegal value, the program should explain why the input is unacceptable, and then the program should repeat the same input question to give the user the opportunity to change the input value. Also, at each input point, the user should be able to abort the current input sequence and return to the main menu.

All single key inputs should provide a single designated escape key for this function. Input-value statements should have a default value, such as 0 or a negative number, that returns the user to the main command menu. Listing 9-2 shows the section of the Takeover program that handles the main control menu.

PROGRAM FLEXIBILITY

As we have said before, the design of your program should be as flexible as possible so that changes to the game design can be made quickly

Listing 9-2. Takeover main control menu.

```
500 REM MAIN MENU
510 GOSUB 900:REM PRINT MENU
520 GOSUB 700:T=T-64
530 IF T<1 OR T>20 THEN 520
540 ONTGO TO 9,4000,4200,1500,550,9,9,9,4300,9,9,9,9,9,85
00,4400,4700,7000,5000
550 REM END TURN
560 GOSUB 3000:REM CLEAR WINDOW
570 PRINT "DO YOU REALLY WISH TO END TURN (Y/N) "
580 GOSUB 700:IF T<>89 THEN 510
585 REM YES END TURN
590 NEXT NP:GOSUB 6000:REM END OF TURN PROCESSING
595 IF NT=TT THEN 645
```

and without major reprogramming of the program's logical or control flow. Rarely will your first attempt at a simulation model or artificial-intelligence algorithm be perfect; therefore, you will find that you constantly need to alter these routines. If each of these changes requires a substantial reprogramming effort, you will never complete your game-design project. The following are a few pointers on how to maintain flexibility in your programs.

Using Variables Rather than Constants for Parameters

By using variables instead of constants in your program, you can initialize the variables in a single place in your program. If you decide a value should be changed, you change the corresponding variable. You are thereby assured of the change throughout your program. Consider the artificial-intelligence algorithm we developed for a computer to determine run/pass defenses in a football simulation:

$$E = 10 Y/D - 7D$$

Let us assume that this formula is to be used many times within our program and that the values 10 and 7 are not the optimum values for the weighting factors in this equation. If we code this equation

$$E = (Y/D * 10) - (7 * D)$$

then in every place it is used in the code, we would have to alter the values 10 and 7. In addition, we would have to be sure we found all the places in the program where this equation is used. If, instead, we initially coded this equation

$$W1 = 10:W2 = 7$$

and used the following equation everywhere else in program

$$E = (W1 * Y/D) - (W2 * D)$$

then by simply changing the values of W1 and W2 in the initializing phase, we could make the change throughout the entire program.

Using Arrays Instead of Multiple Variables

Arrays or subscripted variables are much more flexible than normal variables because you can easily alter the program by simply changing the array dimension. As an extreme example, assume that we programmed the game Prime Time without using array variables. You might have variables such as AR, CR, and NR corresponding to the ratings for ABC, CBS, and NBC, and similar variables for other program values. Now assume that you decided to make Prime Time a game for four players by adding the PBS network. Can you imagine all the reprogramming that this change would require? By

Listing 9-3. Prime Time color setting routine.

```
10000 REM SET COLOR FOR PLAYER
10010 IF NP=1 THEN PRINT "{LRED}";
10020 IF NP=2 THEN PRINT "{CYAN}";
10030 IF NP=3 THEN PRINT "{YELO}";
10040 RETURN
```

using subscripted variables, you would simply change the array dimension from 3 to 4, and a great part of your program would not need to be changed.

Using Subroutines to Perform Repeated Control and Initialization Functions

By using subroutines to perform repeated control and initialization functions, we can be sure there is a single place in our program where we can make a change that will be made globally throughout our program. An example of this is the subroutine (Listing 9-3) from the game Prime Time that controls the color settings.

As you may have noticed if you have played Prime Time, a different color is associated with each of the networks in the game. The colors make the program clearer and more aesthetically pleasing. Each time the program is ready to change the network color, a subroutine is called. If you decide to alter the colors used for each network, you only have to make a simple change to this subroutine.

PROGRAM STRUCTURE

It is important for a program to have a well-

defined structure so control flow is not extremely difficult to follow and the potential impact of coding modifications are not unclear. The following techniques work well for BASIC programs. You must be extremely careful in the use of BASIC, which, with its proliferation of GOTOs, tends to lead to very unstructured code. We suggest that you break the program into three control sections: an initialization section, a main game control loop, and an end-of-game processing section. Each of these sections directs program flow control by branching to other appropriate sections of the program to perform a task and then returns to the control section for further processing. We have found that it is a good practice to use even-numbered lines, 1000, 2000, and so forth, to start each program routine that performs a specific task. This makes the program much easier to document and follow. Global renumbering of program lines might make your listing more pleasing, but we strongly discourage using this practice outside of a single program routine unless you simply must make some space for an added program line.

An example of this structured BASIC approach can be seen in the first 700 lines of Takeover (Listing 9-4).

Listing 9-4. Example of structured BASIC.

```
1 GOTO 100
9 GOTO 520
100 GOSUB 20000:REM INITIALIZE PROGRAM
101 PRINT "{C/DN}{C/DN}{C/DN}DO YOU WISH TO LOAD A SAVED
GAME(Y/N)·":GOSUB700
103 IF T=89 THENE=0:GOSUB9700:GOTO190
110 GOSUB 15000:REM GET GAME OPTIONS
120 GOSUB 10000:REM INITIALIZE CORPORATIONS
190 IF E=1 THEN E=0:GOTO101
```



```

200 FOR NT=FT TO TT
250 GOSUB 1100:REM CALCULATE PRICES
260 IR=IR+INT(RND(0)*4-RND(0)*4):REM CALCULATE INTEREST
RATE
262 IF IR<5 THEN IR=5
264 IF IR>20 THEN IR=20
270 PRINT "{CLR}":GOSUB 2000
300 FOR NP=1 TO TP
310 REM CLEAR TRANSACTION FLAG
320 FOR I=1 TO20:TF(I)=0:NEXT I
450 SF=0
470 GOSUB 3100:SK=NP:FG=0:GOSUB 2500
475 IF SF=1 THEN SK=LC:FG=1:GOSUB 2500
500 REM MAIN MENU
510 GOSUB 900:REM PRINT MENU
520 GOSUB 700:T=T-64
530 IF T<1 OR T>20 THEN 520
540 ONTGO TO9,4000,4200,1500,550,9,9,9,4300,9,9,9,9,9,85
00,4400,4700,7000,5000
550 REM END TURN
560 GOSUB 3000:REM CLEAR WINDOW
570 PRINT "DO YOU REALLY WISH TO END TURN (Y/N)"
580 GOSUB 700:IF T<>89 THEN 510
585 REM YES END TURN
590 NEXT NP:GOSUB 6000:REM END OF TURN PROCESSING
595 IF NT=TT THEN 645
600 PRINT "{CLR}"
610 PRINT "{RVOF}SUMMARY END OF TURN #{RVOF} "NT" OF "TT
620 PRINT:PRINT:GOSUB 11000
630 PRINT:PRINT:PRINT "DO WISH TO SAVE GAME (Y/N)"
635 GOSUB 700:IF T<>89 THEN 640
636 GOSUB 9600:PRINT:PRINT "DO YOU WISH TO END GAME (Y/N)
":GOSUB 700
638 IF T=89 THEN END
640 PRINT:PRINT "CALCULATING NEW MARKET PRICES"
645 NEXT NT
650 PRINT "{CLR}"
660 PRINT "{RVON}END OF GAME SUMMARY{RVOF}"
670 PRINT:PRINT:GOSUB 11000
680 PRINT:PRINT:PRINT"DO YOU WISH TO PLAY AGAIN (Y/N)
690 GOSUB 700
695 IF T=89 THEN 101
699 END
700 REM GET A KEY

```

The initialization section of Takeover is contained in lines 1-200. The main game control loop starts with the statement

```
FOR NT=FT TO TT
```

in line 200 and runs to the NEXT NT in line 645. The end-of-game processing is done in lines 650-699. Also, you can see from the Takeover program documentation chapter that we endeavored to use even-numbered program lines for the major subroutines.

IMPROVING EXECUTION SPEED

If you are programming in BASIC, sooner or later you encounter execution-speed problems. You could choose to accept a rather slow program execution, as was done in the Prime Time program, but at times the execution speed is so slow that it is unacceptable. For example, in the case of the Word Square computer artificial-intelligence chapter, we were forced to code the computer's word-find routine in assembly code. If you encounter program execution problems, the first thing you should do is check your BASIC coding and assure yourself that there are no gross inefficiencies such as useless or unneeded program loops. Then you should review your simulation algorithms and determine whether or not you could simplify them without losing the important factors of the simulation. Should these fixes prove inadequate, you then have two different courses for further action. First, you can rewrite your BASIC code, taking advantage of a number of programming tricks to speed up program execution. Second, you can recode the most time-crucial subroutines in assembly code.

We strongly recommend the second course of action. If you have not yet learned assembly code, you should not be too terrified. While we strongly recommend not using assembly code to write an entire program, it is an excellent partner to BASIC to help out in those small time-crucial routines. Such routines are not that difficult to write and can improve program execution speed by a much greater magnitude than BASIC programming tricks can. When we first started programming Word Square,

it was our intent to write it entirely in BASIC so that it would be in keeping with one of the themes of this book. The first attempt at coding the computer word-find routines in BASIC resulted in a 10-minute execution time for the computer to determine all five words in the square! After considerable reprogramming and using every possible program trick for speed execution, the computer still took about three minutes to find its five words. In the end we had to give in and program the word-find routine in assembly code. The payoff for the relatively simple assembly language routine (less than 200 bytes) is that the computer now determines its five words in less than five seconds, which is more than adequate for play of the game. You should refer to the source coding of this routine presented previously. If you can learn enough assembly language programming to write an equivalent routine, you can solve most, if not all, of the program speed problems you might encounter in BASIC.

SAVING RAM

In most applications, you do not have to be overly concerned about saving computer RAM. However, should you run into a need to save space in your programs, you can try one or more of the following:

1. Remove REM statements. These are important for program documentation but are not used by BASIC during program execution. If you have a printer, you can keep a copy of the program listing with handwritten comments for future reference.

2. Remove spaces. Spaces make program listings more readable, but they are ignored by the interpreter and are not required.

3. Put multiple statements on the same line. For example,

```
10 FOR I = 1 TO 10
20 POKE A + I, O
30 NEXT I
```

can be written

```
10 FOR I=1 TO 10:POKE A+I,O:NEXT
```

4. Replace often-used constants and strings with variables. For example, replace

```
10 ?"HIT ANY KEY"  
20 POKE 50000,0
```

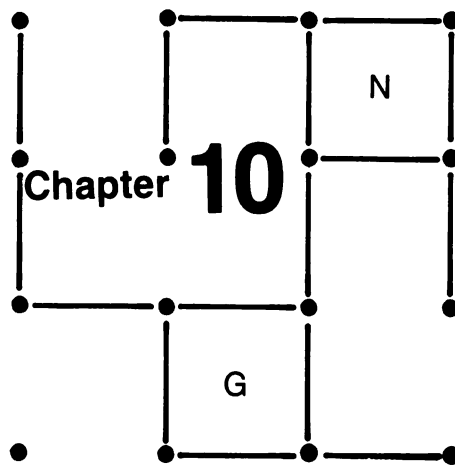
with the lines

```
10 A$="HIT ANY KEY":S=50000  
20 ?A$:POKE S,0
```

5. Replace repetitive sequences with subroutines. If you use a particular sequence of pro-

gram lines repeatedly in your program, you can replace them with a GOSUB to a subroutine that performs the same function.

If you keep in mind the programming considerations we have described, the entire design process, from developing the rough draft program to refining the finished product, runs smoothly. It might be instructive for you to see how we did and did not follow our own advice in the development of the games Prime Time, Word Square, and Takeover, which are fully documented in Chapters 11, 12, and 13.



Implementing a Game Design- Professional Example Games

We have done a great deal of talking about game designing, so now let's show some results. You should have a good idea of what makes a good game, how to approach a game-design project on a microcomputer, how to design simulation models, and how to develop artificial-intelligence algorithms. You should also understand how programming considerations affect the implementation of a game design on a microcomputer. We believe that the best way to understand the material presented in the previous chapters is to study some sample programs.

USING THE FOLLOWING THREE CHAPTERS

The next three chapters contain a thorough documentation of the games Prime Time, Word Square, and Takeover. The chapters include variable and subroutine listings, flowcharts, and the actual BASIC program listings. One of the major objectives of this book is to present the reader with comprehensive examples of complete game-design implementations on a microcomputer. The sample

programs that follow are of equal complexity with typical microcomputer games that sell for \$20 to \$30. The samples are much more complex than the average game program that is usually provided in book format. The manner in which you use these programs depends on your reasons for reading this book. So that you might gain the maximum benefit from the examples, we offer some suggestions for using these next chapters.

If you wish only to play the games, you can simply type in the programs from the listings and enjoy. You will find that the games themselves easily justify the price of this book. If you purchase the loader disk, you can save the typing task, and you can skip the next three chapters. For those of you who are also interested in learning more about the game-design process, perhaps so that you can develop your own game-design projects, further study of these examples should be quite rewarding. Compare the elements of a good game with those in each of the sample programs. What are the elements of these examples that make them good games? How are these elements specifically

achieved in the program? What negative aspects did you find in these games? How could the designs have been changed to correct these failings? (No game is perfect; there is always room for improvement.) Study how the major simulation models and artificial-intelligence algorithms were implemented in each of these programs. What compromises were made in the design of these models? Why were they made? Could they be implemented in a better or more effective way?

If you are interested in computer programming as well as game design, study the actual BASIC coding for each game. Review the chapter on programming considerations and study the structure of each program. Look at the subroutine list and flowcharts to see how tasks are subdivided in the program. What compromises were made because the game is being implemented on a microcomputer? Are these compromises necessary, or could they have been avoided by changing the design of the game? How is program performance affected by the choice of BASIC? How and where could these programs be improved by using machine language?

If you are really interested in designing your own microcomputer game, you can use these programs as a framework for perfecting your game-design techniques prior to creating a design of your own. Test your ideas for making each of these games better. Try different simulation models that might be more efficient or realistic. Alter the artificial-intelligence routines and see how computer play and

execution speed are affected. If you know machine language, you could rewrite part of the coding and observe the effect on program execution speed. The possibilities are endless, so have a good time and do not be afraid to dabble with our coding.

GAME-DESIGN PROJECTS

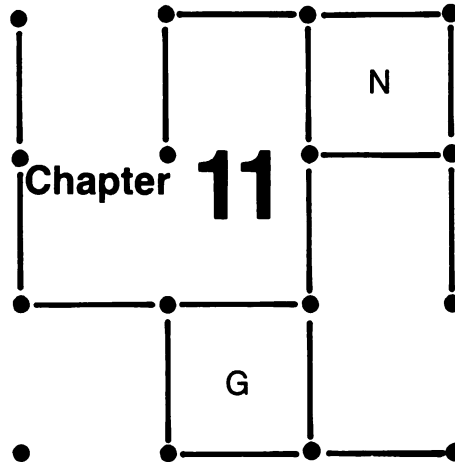
For you really serious game-design students, we offer the following projects:

1. Rewrite in machine language the Prime Time routine that sorts the shows in order of their power ratings. You need to set up a routine that passes the values of the current rating to the machine code by show number and returns the show numbers to BASIC in the order of their rating.

2. Add some creative sound and graphics to the Word Square program that increase the visual appeal and player enjoyment of the game. You can add your own title page. You can display a graphic intermission between turns so that the players can relax as they get ready to strain their brains during the next turn. You could add a graphic display at the end of the game to show the players' ratings.

3. Develop the artificial-intelligence routines that let the computer play one of the major corporations in the Takeover game.

These suggested projects should keep you busy for awhile. Now it is time for Prime Time.



Prime Time Program Documentation

This chapter gives a detailed description of the program Prime Time. First, we define the use of all variables, arrays, and subroutines used in the program. We follow this with a very simple flowchart for the entire program (Fig. 11-1) and go through the main program flow line by line, highlighting the subroutines and their functions. This is followed at the end of the chapter by an actual listing of the entire program (Listing 11-1).

PRIME TIME VARIABLES

The following variables are used in Prime Time:

TR—Total rating used by computer to determine its optimal show lineup.

TP—Total power of all three shows in a given day slot.

L—Loop counter.

W—Week number.

NT—Number of turns in the game.

K—Loop counter used to loop through networks (1-3).

NP—Number of the network currently playing.

MS—Maximum number of swaps the computer performs during its lineup selection phase.

E—Error condition; 1 = error detected.

I\$—String variable used to accept keyboard input.

MX—Maximum rating value; used to determine winning player.

MI—Network number that corresponds to the MX value.

I—Loop counter.

T—Temporary value storage.

V—Temporary value storage.

F—Flag used to swap colors.

J—Loop counter.

JJ—Flag used by computer in lineup-selection routine.

V1—Temporary value storage.

V2—Temporary value storage.

SP—Flag used during sorting of the ratings list (if SP=1, then swap was made).

SM—Holds sum during ratings calculations.

SS—Loop counter.

MY—Money left during production and promotion phase.

SN—Show number to which random event occurs.

GB—Random event number.

PRIME TIME ARRAY VARIABLES

The numbers in parentheses are the array dimensions of the variables.

RT (21)—Show rating indexed by show number.

PW (21)—Show power rating indexed by show number.

OW (21)—Network (1-3) that owns the show indexed by show number.

BD (3,7)—Board array containing show numbers. The first index is the network number. The second index is the number of the day of the week.

TB (3,7)—Temporary board use in the computer routine to determine lineup. It uses the same indexes as in BD.

RL (21)—Ratings list that contains show numbers. It is indexed by rating number.

SW\$ (21)—Contains show names. It is indexed by show numbers.

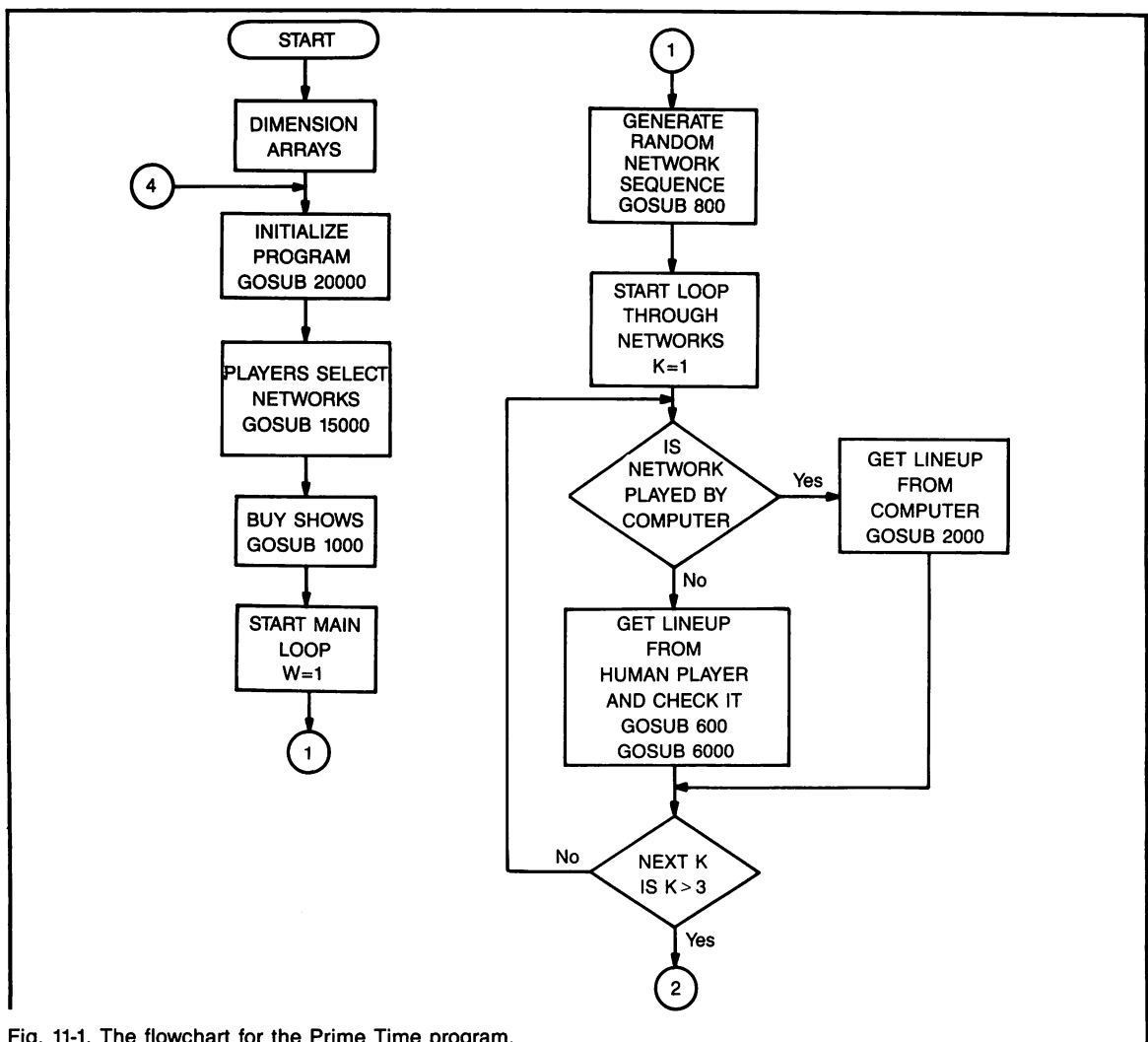
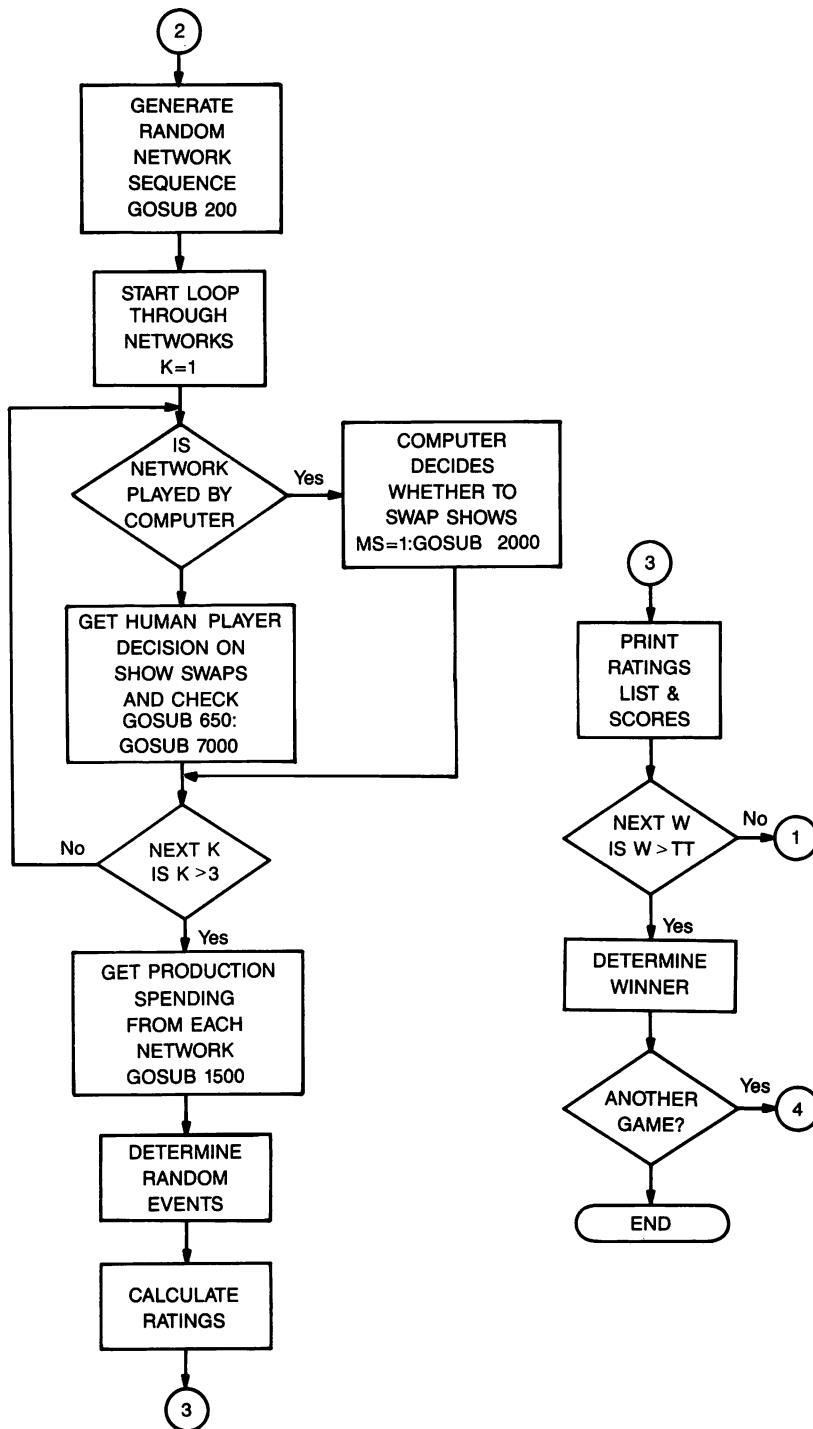


Fig. 11-1. The flowchart for the Prime Time program.




```

1 REM PRIME TIME
2 REM BY GEORGE SCHWENK
3 REM (C) 1984 TAB BOOK CO.
4 REM ALL RIGHTS RESERVED
5 PRINT "{CLR}{C/DN}{C/DN}{C/DN}{C/DN}"TAB(15)"{RVON}PRIME
   TIME{RV OF}"
6 PRINT"{C/DN}"TAB(19)"BY":PRINTTAB(13)"GEORGE SCHWENK":
   PRINT"{C/DN}{C/DN}"
7 PRINT TAB(10)"(C) 1984 TAB BOOK CO."
8 PRINT:PRINTTAB(11)"ALL RIGHTS RESERVED"
9 GOTO 100
10 REM CALCULATE TR FOR NP FROM TB
20 TR=0
30 FOR L=1 TO 7
40 TP=PW(TB(1,L))+PW(TB(2,L))+PW(TB(3,L))
50 TR=TR+PW(TB(NP,L))*VW(L)/TP
60 NEXT L:RETURN
100 REM DIMENSION ARRAYS
110 DIM RT(21),PW(21),OW(21),BD(3,7),TB(3,7),RL(21),SW$(21)
120 DIM SH(21),EV$(21)
121 DIM AD(21),AV(3,7)
130 GOSUB20000:REM INITIALIZE
140 GOSUB 15000:REM NETWORK SELECTION
150 GOSUB1000:REM BUY SHOWS
200 REM MAIN GAME LOOP W=WEEK#
210 FORW=1TONT
220 GOSUB800:REM GENERATE SEQ
230 FORK=1TO3:REM PLAYER LOOP"
240 NP=SQ(K)
250 IF PY(NP)=0 THEN MS=7:GOSUB2000:GOTO350
260 GOSUB5000:REM PRINT BOARD
265 GOSUB 4000:REM PRITN SHOWS OWNED
270 PRINT "ENTER WEEKLY LINEUP - 7 LETTERS"
280 GOSUB 600
290 GOSUB 6000:REM CHECK INPUT STRING
300 IF E=1 THEN 260
310 GOSUB 5000
320 PRINT:PRINT "IS LINEUP OK (Y/N)?"
330 GOSUB 650
340 IF I$<>"Y" THEN 260
350 NEXT K
360 GOSUB 800
370 FORK=1 TO 3
380 NP=SQ(K)
390 IF PY(NP)=0 THEN MS=1:GOSUB2000:GOTO480

```

```

400 GOSUB 5000
410 PRINT:PRINT "DO YOU WISH TO SWAP TWO SHOWS (Y/N)?"
420 GOSUB 650
430 IF I$<>"Y" THEN 480
440 PRINT:PRINT "ENTER DAY #'S FOR SHOWS (1-7)"
450 GOSUB 600
460 GOSUB7000:REM CHECK IF SWAP IS OK
470 IF E=1 THEN PRINT:PRINT "SWAP NOT ALLOWED":GOSUB700
    :GOTO400
480 NEXTK
481 REM ASK FOR P&P
482 GOSUB 800:FORK=1TO3
483 NP=SQ(K):GOSUB 1500
485 NEXT K
486 FORK=1 TO3
487 GOSUB1700:GOSUB710:GOSUB710:REM RANDOM EVENT
488 NEXTK
490 GOSUB 8000:REM CALC RATING & SORT
495 GOSUB 9000:REM PRINT RATINGS LIST
499 NEXT W
500 PRINT:PRINT:PRINT,"END OF GAME"
510 PRINT"(CLR)"
520 MX=0:MI=1
530 FOR I=1 TO 3
540 IF YR(I)>MX THEN MX=YR(I):MI=I
550 NEXT I
560 PRINT NT$(MI); " WINS !!!!!"
570 PRINT:PRINT "DO YOU WISH TO PLAY AGAIN (Y/N)?"
580 GOSUB 650:IF I$="Y" THEN 130
590 REM RETURN TO BASIC
599 END
600 REM GET INPUT
610 PRINT NT$(NP);
620 INPUT I$
630 RETURN
650 REM GET Y/N
660 PRINT NT$(NP); ">";
670 GOSUB740:PRINT I$
680 T=ASC(I$)
690 RETURN
700 REM DELAY SUBROUTINE
710 FOR I=1TO3000
720 NEXT
730 RETURN
740 REM GET KEY
750 GET I$:IF I$="" THEN 750

```

```

760 RETURN
800 REM GENERATE A RANDOM SEQUENCE FROM 1 TO 3
810 T=1:SQ(1)=0:SQ(2)=0:SQ(3)=0
820 V=INT(RND(0)*3+1)
830 IF V=SQ(1) OR V=SQ(2) OR V=SQ(3) THEN 820
840 SQ(T)=V:T=T+1
850 IFT<4 THEN 820
860 RETURN
900 REM LIST SHOWS AND OWNERS
910 PRINT CHR$(147):F=0
920 PRINT "{LRED}          <<<      SHOW      >>>"TAB(28)"POWER"
      TAB(34)"OWNER"
930 FOR I=1 TO 18:F=NOT(F)
935 IF F=0 THEN PRINT "{LBLU}";:GOTO 940
936 PRINT "{CYAN}";
940 PRINT SW$(I)TAB(28)PW(I);
950 IFOW(I)=0 THEN PRINT TAB(35)"NONE":GOTO 970
960 PRINT TAB(35)NT$(OW(I))
970 NEXT
980 RETURN
1000 REM GENERATE CHOICE
1010 FORJ=1TO6
1020 GOSUB 800:REM GET RANDOM SEQ
1030 FORK=1TO3:NP=SQ(K)
1040 IF PY(NP)=0 THEN GOSUB3000:GOTO1110
1050 GOSUB 900:REM LIST SHOWS
1060 PRINT"PRESS LETTER TO SELECT;   MONEY=";PT(NP)
1070 GOSUB650
1080 V=T-64:IFV<1 OR V>18 THEN1050
1090 IFOW(V)<>0 THEN PRINT:PRINT "ALREADY OWNED - TRY
      AGAIN":GOSUB 700:GOTO1050
1110 IFPW(V)>PT(NP)THENPRINT:PRINT"NOT ENOUGH MONEY - TRY
      AGAIN":GOSUB700:GOTO1040
1120 PT(NP)=PT(NP)-PW(V)
1130 OW(V)=NP
1140 PRINT:PRINT NT$(NP)" SELECTS "SW$(V)
1150 GOSUB 700:NEXTK:NEXTJ
1160 REM SET NEW SHOWSFOR LAST TURN
1170 OW(19)=1:PW(19)=PT(1)
1180 OW(20)=2:PW(20)=PT(2)
1190 OW(21)=3:PW(21)=PT(3)
1200 REM SETUP BOARD FOR FIRST TURN
1210 IP(1)=1:IP(2)=1:IP(3)=1
1220 FORI=1TO21
1230 BD(OW(I),IP(OW(I)))=I
1240 IP(OW(I))=IP(OW(I))+1

```

```

1250 NEXT I
1260 RETURN
1500 REM ROUTINE FOR ALLOCATING P&P
1504 MY=100:L=0
1505 IF PY(NP)=0 THEN 1530
1510 PRINT CHR$(147)
1520 PRINT "(RVON)PRODUCTION AND PROMOTION(RVOF)":GOSUB700:
    GOSUB5000
1530 FORP=1TO21
1535 IF MY=0 THEN 1600
1540 IF OW(P)<>NP THEN 1600
1542 IF L=3 AND PY(NP)=1 THEN GOSUB 5000
1545 IF L=6 THEN V=MY:GOTO 1590
1546 IFPY(NP)=0 THEN V=INT(RND(O)*MY):GOTO1590
1550 PRINT:PRINT "ENTER $$$ TO SPEND ON P&P FOR"
1560 PRINT SW$(P)TAB(28)"MONEY= ";MY
1570 GOSUB600:V=VAL(I$)
1580 IF V>MY THEN PRINT "NOT ENOUGH MONEY":GOTO 1550
1590 MY=MY-V:L=L+1:GOSUB13000:AV(NP,S)=INT(V/3)
1600 NEXT P:RETURN
1700 REM RANDOM EVENT
1705 PRINT CHR$(147)
1710 SN=INT(RND(O)*21+1)
1720 GB=INT(RND(O)*21+1)
1730 PW(SN)=PW(SN)+AD(GB)
1740 IF PW(SN)<1 THEN PW(SN)=1
1750 PRINT "NEWS FROM SET OF ";SW$(SN)
1760 PRINT EV$(GB)
1770 PRINT:PRINT"THE POWER RATING OF THE SHOW IS"
1780 IF AD(GB)<1 THEN 1800
1790 PRINT " INCREASED BY ";AD(GB);"POINTS":RETURN
1800 PRINT " DECREASED BY ";ABS(AD(GB));"POINTS":RETURN
2000 REM COMPUTER DECISION
2005 PRINT:PRINT NT$(NP);" IS THINKING VERY HARD"
2010 IF W>1 THEN 2100
2020 REM USE SELECTION SEQUENCE
2030 J=1:FOR I=1 TO 21
2040 IF OW(I)=NP THEN BD(NP,J)=I:J=J+1
2050 NEXT I
2055 SP=0:REM SORT SHOWS BY POWER
2060 FOR J=1 TO 6
2065 IF PW(BD(NP,J))>=PW(BD(NP,J+1)) THEN 2075
2070 T=BD(NP,J):BD(NP,J)=BD(NP,J+1):BD(NP,J+1)=T:SP=1
2075 NEXT J
2080 IF SP=1 THEN 2055
2100 REM SET TB=BD

```

```

2110 FOR J=1 TO 7
2120 FOR I=1 TO 3
2130 TB(I,J)=BD(I,J)
2140 NEXT I:NEXT J
2200 REM GO THROUGH LINEUP
2210 REM CALCULATE BASE VALUE
2220 GOSUB 10
2230 MR=TR
2235 JJ=0
2240 J=1:SP=0
2250 REM TRY A SWAP
2252 K1=J:K2=J+1:IF J=7 THEN K2=1
2260 V1=TB(NP,K1):V2=TB(NP,K2)
2270 TB(NP,K2)=V1:TB(NP,K1)=V2
2280 GOSUB 10:REM CALCULATE TR
2290 IF TR>MR THEN SP=SP+1:MR=TR:GOTO 2320
2300 REM SWAP IS NOT BENEFICIAL RESTORE BOARD
2310 TB(NP,K1)=V1:TB(NP,K2)=V2
2320 IF SP=MS THEN 2340
2330 J=J+1:IF J<8 THEN 2250
2335 IF JJ=0 THEN JJ=1:GOTO 2240
2340 REM SET BD=TB
2350 FOR J=1 TO 7
2360 BD(NP,J)=TB(NP,J)
2370 NEXT J
2380 RETURN
3000 REM COMPUTER BUYS SHOWS
3010 V=INT(RND(0)*18+1)
3020 IF OW(V)<>0 THEN 3010
3030 IF PW(V)>PT(NP)-1 THEN 3010
3040 RETURN
4000 REM LIST SHOWS OWNED BY NP
4005 GOSUB 10000
4010 PRINT:PRINT "SHOWS OWNED BY "NT$(NP)
4020 FOR I=1 TO 21
4030 IF OW(I)<>NP THEN 4050
4040 PRINT SW$(I)
4050 NEXT I
4060 PRINT "{LBLU}":RETURN
5000 REM PRINT BOARD
5010 PRINT CHR$(147)
5020 PRINT " {RVON}#    DAY VW ABC   CBS   NBC WK #{RVOF}"W
5030 FOR I=1 TO 7
5040 PRINT I;DY$(I)TAB(12);VW(I);
5050 PRINT CHR$(BD(1,I)+64)PW(BD(1,I));
5060 PRINT TAB(21)CHR$(BD(2,I)+64)PW(BD(2,I));

```

```

5070 PRINT TAB(26)CHR$(BD(3,I)+64)PW(BD(3,I))
5080 NEXT I:RETURN
6000 REM CHECK 7 LETTER SEQ
6010 E=0
6015 IF LEN(I$)=1 THEN RETURN
6020 IF LEN(I$)<>7 THEN PRINT "EXACTLY 7 LETTERS PLEASE":
    GOTO 6200
6030 REM CHECK OWNERSHIP
6040 FOR I=1 TO 7:L=ASC(MID$(I$,I,1))-64
6045 IF L<1 OR L>21 THEN E=1:GOTO 6060
6050 IF OW(L)<>NP THEN E=1
6060 NEXT I
6070 IF E=1 THEN PRINT "YOU MUST OWN ALL SHOWS ENTERED":
    GOTO 6200
6080 REM CHECK IF SHOW IS REPEATED
6090 FOR I=1 TO 6
6100 FORJ=I+1TO7
6110 IF MID$(I$,I,1)=MID$(I$,J,1) THEN E=1
6120 NEXTJ:NEXTI
6130 IF E=1 THEN PRINT "YOU MUST ENTER 7 DIFFERENT SHOWS":
    GOTO 6200
6140 REM NO ERRORS MAKE IT HERE
6150 FOR I=1 TO 7
6160 BD(NP,I)=ASC(MID$(I$,I,1))-64
6170 NEXT I
6180 RETURN
6200 E=1:GOSUB 700:RETURN
7000 REM CHECK IF SWAP IS OK
7010 IF LEN(I$)<>2 THEN E=1:RETURN
7020 E=0:V1=VAL(MID$(I$,1,1)):V2=VAL(MID$(I$,2,1))
7030 IFV1<1 OR V1>7 OR V2<1 OR V2>7 THEN E=1
7040 IF V1=V2 THEN E=1
7050 IF E=1 THEN RETURN
7060 REM MAKE SWAP
7070 V=BD(NP,V1)
7080 BD(NP,V1)=BD(NP,V2)
7090 BD(NP,V2)=V
7100 RETURN
8000 REM CLACULATE RATINGS
8005 PRINT CHR$(147):PRINT"<<< PLEASE WAIT >>>":PRINT:
    PRINT"CALCULATING RATINGS FOR WEEK #";W
8010 FOR J=1 TO 7:REM DAY LOOP
8020 TP=PW(BD(1,J))+PW(BD(2,J))+PW(BD(3,J))+AV(1,J)+
    AV(2,J)+AV(3,J)
8030 FORI=1 TO 3:REM NETWORK LOOP
8035 SH(BD(I,J))=100*(PW(BD(I,J))+AV(I,J))/TP

```

```

8040 RT(BD(I,J))=VW(J)*(PW(BD(I,J))+AV(I,J))/TP
8050 NEXT I:NEXT J
8060 REM SORT RLIST
8070 SP=0
8080 REM GO THROUGH LIST
8090 FOR I=1 TO 20
8100 IF RT(RL(I+1))>RT(RL(I)) THEN SP=1:T=RL(I):RL(I)=
      RL(I+1):RL(I+1)=T
8110 NEXT I
8120 IF SP=1 THEN 8070
8130 REM SET POWER
8140 FOR I=1 TO 21
8150 PW(RL(I))=30-I
8160 NEXT I
8170 RETURN
9000 REM LIST TOP SHOWS
9010 PRINT CHR$(147)
9020 PRINT "{RVON}TOP SHOWS FOR WEEK #";W
9030 PRINT "{RVON}RANK          SHOW          RATE   %"
      :GOSUB 700
9040 FOR I=1 TO 21
9050 NP=OW(RL(I)):GOSUB 10000:REM SET NETWORK COLOR
9070 PRINT ITAB(4)SW$(RL(I))TAB(31)INT(RT(RL(I)))
      TAB(35)INT(SH(RL(I)))
9080 NEXT I
9090 REM CALCULATE WEEK AND YEAR RATINGS
9100 FOR J=1 TO 3
9110 SM=0
9120 FOR I=1 TO 7
9130 SM=SM+RT(BD(J,I))
9140 NEXT I
9150 WK(J)=SM/7
9160 YR(J)=(YR(J)*(W-1)+WK(J))/W
9170 NEXT J
9175 PRINT CHR$(154)
9180 GOSUB 9200
9190 GOSUB 740:RETURN
9200 REM PRINT OUT RATINGS
9210 PRINT "{RVON}RATING",
9211 NP=1:GOSUB 10000:PRINT" ABC ",
9212 NP=2:GOSUB 10000:PRINT" NBC ",
9213 NP=3:GOSUB 10000:PRINT" CBS ",
9220 PRINT "{LBLU}{RVON}WEEK{RVDF}",
9222 I$=STR$(WK(1)):PRINT LEFT$(I$,5),
9224 I$=STR$(WK(2)):PRINT LEFT$(I$,5),
9226 I$=STR$(WK(3)):PRINT LEFT$(I$,5)

```

```

9230 PRINT "{RVON}YEAR{RVDF}",
9232 I$=STR$(YR(1)):PRINT LEFT$(I$,5),
9234 I$=STR$(YR(2)):PRINT LEFT$(I$,5),
9236 I$=STR$(YR(3)):PRINT LEFT$(I$,5);
9240 RETURN
10000 REM SET COLOR FOR PLAYER
10010 IF NP=1 THEN PRINT "{LRED}";
10020 IF NP=2 THEN PRINT "{CYAN}";
10030 IF NP=3 THEN PRINT "{YELO}";
10040 RETURN
13000 REM RETURN IN S LINEUP SPOT OF PROG P FOR NP
13010 FOR SS=1 TO 7
13020 IF BD(NP,SS)=P THEN S=SS
13030 NEXT SS:RETURN
15000 REM NETWORK SELECTION
15010 PRINT CHR$(147)
15020 PRINT "<< NETWORK SELECTION PHASE >>"
15030 FORI=1TO3
15040 PY(I)=0:NP=I
15050 PRINT"DOES A HUMAN WISH TO PLAY ";NT$(I);" ?"
15060 PRINT "ENTER (Y/N)":GOSUB 650
15070 IF I$="Y" THEN PY(I)=1
15080 PRINT:PRINT:NEXTI
15085 PRINT "ENTER # OF WEEKS YOU WISH TO PLAY";:
      INPUT I$:NT=VAL(I$)
15090 PRINT CHR$(147):PRINT TAB(10)"<<< SELECTIONS >>>"
15100 PRINT:FORI=1TO3
15110 PRINT NT$(I);
15120 IF PY(I)=0 THENPRINT" COMPUTER":GOTO15140
15130 PRINT" HUMAN"
15140 PRINT
15150 NEXTI
15155 PRINT"# OF WEEKS= ";NT:PRINT
15160 PRINT"ARE THESE OK? (Y/N)"
15170 GOSUB 740
15180 IF I$<>"Y" THEN 15000
15190 RETURN
20000 REM INITIALIZE PROGRAM
20010 REM *****
20020 PRINT "{C/DN}{C/DN}<<<PLEASE WAIT - INITIALIZING>>>"
      :GOSUB 710
20030 REM SET SHOW DATA
20040 RESTORE:FW(0)=0
20050 FOR I=1TO21
20060 READ SW$(I),FW(I)
20070 DW(I)=0

```



```

20080 NEXT I
20110 REM SET DAY DATA AND CLEAR BOARD
20120 FOR I=1 TO 7
20130 READ VW(I)
20140 BD(1,I)=0
20150 BD(2,I)=0
20160 BD(3,I)=0
20170 NEXT I
20200 REM SET NET WORK DATA
20210 FOR I=1 TO 3
20220 PT(I)=126
20230 WK(I)=0
20240 YR(I)=0
20250 NEXT I
20260 REM READ IN DAY AND NETWORK STR
20270 FOR I=1 TO 7
20280 READ DY$(I):NEXT I
20290 FOR I=1 TO 3
20300 READ NT$(I):NEXT I
20310 REM SET RL AND READ ADDER
20320 FOR I=1 TO 21
20330 RL(I)=I
20340 READ AD(I)
20350 NEXT I
20360 REM READ IN EVENTS
20370 FOR I=1 TO 21
20380 READ EV$(I):NEXT I
20400 REM SET DEFAULT COLORS
20410 POKE 53280,2:POKE 53281,6
20999 RETURN
30000 REM SHOW DATA
30010 DATA ALL IN LA FAMILIA ,25
30020 DATA BRANANZA ,25
30030 DATA CAGNEY & LAGNEY ,25
30040 DATA DIE NASTY ,25
30050 DATA E TEAM ,20
30060 DATA FALCON QUEST ,20
30070 DATA GOODNIGHT AMERICA ,20
30080 DATA HAWAII O/5 ,20
30090 DATA ILL STREET BLUES ,20
30100 DATA JOHN E. CARSON ,20
30110 DATA KOJERK ,15
30120 DATA LANDING KNOTS ,15
30130 DATA MAGNIFICENT P.I. ,15
30140 DATA NIGHT RIDING ,15
30150 DATA OWING MARSHALL ,15

```

```

30160 DATA PEYTON PLAYPEN , 10
30170 DATA QUINN SEES , 10
30180 DATA REMINTON IRON , 10
30190 DATA SOAPY , 10
30200 DATA TRAPPING JOHN M.D. , 10
30210 DATA UNHAPPY DAYS , 10
30220 REM VIEWER #'S
30225 DATA 75, 70, 65, 65, 60, 55, 50
30250 DATA SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY
30260 DATA "{RVON}ABC{RVOF}", "{RVON}CBS{RVOF}",
"{RVON}NBC{RVOF}"
30330 REM ADDER'S
30340 DATA -3, -10, -3, -5, -5, -10, 5, 10, 3, 3, 5, 7, -7, -5, -7,
-5, -3, -3, 10, 3, 7
30400 REM EVENTS
30410 DATA THE FEMALE CO-STAR TELLS BIG LIE ABOUT HER AGE
30420 DATA THE FEMALE STAR QUILTS SHOW TO BECOME A NUN
30430 DATA THE LEADING MAN IS DISCOVERED WEARING A TOUPEE
30440 DATA THE LEADING MAN HAS GAINED 25 POUNDS
30450 DATA THE MALE STAR CATCHES A SOCIAL DISEASE
30460 DATA THE MALE STAR RUNS AWAY WITH BOYFRIEND
30470 DATA THE MALE STAR POSES FOR HIT NUDE POSTER
30480 DATA THE LEADING MAN WINS AN EMMY FOR BEST ACTOR
30490 DATA THE CO-STAR WRITES A BEST SELLING NOVEL
30500 DATA THE LEADING LADY APPEARS ON POPULAR TALK SHOWS
30510 DATA THE LEADING LADY HAS SILICON IMPLANTS
30520 DATA THE LEADING LADY POSES FOR BEST SELLING CALENDER
30530 DATA THE FEMALE STAR GETS PREGNANT
30540 DATA THE FEMALE CO-STAR BREAKS HER LEG
30550 DATA THE WRITERS HAVE WRITER'S BLOCK
30560 DATA THE CAMERAMEN GO ON STRIKE
30570 DATA FAMOUS STUNTMAN IS SERIOUSLY INJURED ON THE SET
30580 DATA A FIRE HALTS PRODUCTION
30590 DATA SHOW WINS EMMY FOR BEST SERIES
30600 DATA SHOW GETS NEW DIRECTOR
30610 DATA SHOW HITS #1 IN JAPAN MEXICO & ENGLAND

```

SH (21)—Contains the percentage of the audience share. It is indexed by show number.

EV\$ (21)—Contains the text of event happenings. It is indexed by a random-event number.

AD (21)—Value to add to power rating when an event occurs. It is indexed by the random-event number.

AV (3,7)—Contains power-rating increase due

to promotion and production expenditures. It has the same indexes as BD.

SQ (3)—Contains network numbers. It is indexed by the K loop counter.

PY (3)—Contains a flag. If equal to zero, then the computer plays this network. It is indexed by network number.

WK (3)—Contains the weekly rating of all

shows owned by a network. It is indexed by network number.

YR (3)—Contains the cumulative, or yearly, rating of all shows owned by a network. It is indexed by network number.

NT\$ (3)—Contains a network name. It is indexed by network number.

PT (3)—Contains the amount of money left during the show-purchase phase. It is indexed by network number.

VW (7)—Contains the number of viewers in a time slot. It is indexed by day number.

PRIME TIME SUBROUTINES

The following subroutines are used in Prime Time:

10—This subroutine calculates the rating total (RT) for the lineup currently in TB for network number NP. It is used by the computer to evaluate its lineup.

600—Accepts line input from players.

650—Accepts single key input from players.

700—Delay subroutine.

800—Generates a random order of the numbers 1, 2, and 3 and stores them in array variable SQ.

900—Lists all shows and owners.

1000—Show-selection subroutine.

1200—Sets up the playing board (BD) for the first turn.

1500—Handles player-spending allocations for production and promotion.

1700—Generates and displays random events.

2000—Computer decides on lineup or swaps for this turn. On entry, MS contains the maximum number of swaps the computer performs to determine the optimal lineup. On exit, BD contains the computer's show lineup for this turn.

3000—Computer decides which show to buy. On exit, V contains the number of the show the computer wants to purchase.

4000—Lists shows owned by network number NP.

5000—Prints main playing board.

6000—Checks a player's seven-letter sequence

designating the desired lineup. On entry, I\$ contains the seven-letter sequence. On exit, E=1 if an error is detected in the sequence.

7000—Checks a proposed show swap. On entry, I\$ contains swap numbers. On exit, E=1 if an error is detected.

8000—Calculates ratings. This routine sets the arrays RT, SH, and RL.

9000—Lists the shows in ratings order, giving weekly rating and percentage of audience share.

10000—Sets the color for player number NP.

13000—Determines the lineup spot of show number P, which is owned by network number NP. On exit, S equals the day number of the show.

15000—Players select networks and play options.

20000—Reads in program data and initializes program variables and arrays.

PRIME TIME PROGRAM DESCRIPTION

The main flow of the Prime Time program begins at line 100. Lines 100 to 200 comprise the initialization portion of the program. First, the array variables are dimensioned and then all variables are initialized by a call to the subroutines at line 20000. This is followed by subroutine calls to line 15000 to view game options and to line 1000 to allow players to purchase their shows. The main control loop for the program begins at line 200 where the week number loop counter (W) is set to run from 1 to NT, which is the number of turns selected during the game option phase previously processed by the subroutine at line 15000.

The main control loop (lines 200-499) consists of three subphases. For each subphase a random sequence of the numbers 1, 2, and 3 is generated by calling the subroutine at line 800. This sequence is used to determine the order in which the networks play during each subphase of the game. During the first subphase, players choose their show lineup for the week. If the flag PY(NP)=0, it indicates that network NP is being played by the computer. A call is made to the subroutine at line 2000, where the computer selects a programming lineup for this week. If the flag PY(NP)=1, it indicates that network NP is being played by a human player. If this

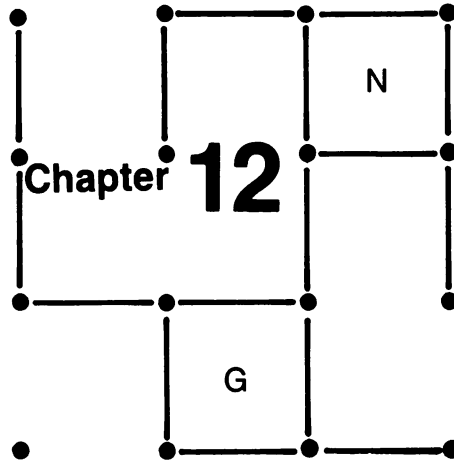
is the case, the board is printed by a GOSUB 5000. Player NP's shows are printed by a GOSUB 4000 and the player's lineup sequence is accepted by a GOSUB 600. This sequence, which is now contained in the variable I\$, is checked with a GOSUB 6000. Upon exit from this subroutine, the flag E is checked. If E=1 it indicates that an error was detected in the player's proposed lineup, and the input sequence is repeated. Otherwise the player is asked if the lineup as entered is okay; this gives the option to reenter the lineup. If the player is satisfied with the lineup, the loop counter K is incremented and play passes to the next network contained in the sequence SQ.

The second subphase of each week's turn runs through a sequence very similar to that in the first subphase. It should be noted that if $PY(NP)=0$, then within this subphase the variable MS is set equal to 1 before the computer is asked for the lineup sequence by a GOSUB 2000. By setting MS=1, the computer performs only one swap before returning. This convention allows the same artificial-intelligence routines to be used by the computer to determine both the initial lineup and the show swap decision.

During the third subphase of each week's turn, players determine spending for production and promotion. This is handled by the subroutine at line 1500, which constructs the array AV that contains the power rating increase to be applied to each show because of production and promotion spending.

Following these three subphases, three random events are generated and displayed by successive calls to the subroutine at line 1000. Each turn is ended with a GOSUB 8000, which calculates the ratings for each show and sorts them from the highest- to lowest-rated show. Finally, a GOSUB 9000 prints the list.

When the loop counter W is greater than NT, the game is over and the main control loop is exited. The winner is determined by comparing each network's cumulative, or yearly, rating (YR). This comparison loop uses the variables MX to store the highest YR, and MI to store the corresponding network number. Finally, after the winner is displayed, the player(s) are asked if they want to play another game. If they do, control is passed to the beginning of the program initialization phase at the point right after the arrays are dimensioned. (This prevents an array-already-dimensioned error.)



Word Square Program Documentation

This chapter gives a detailed description of the program Word Square, which actually contains two programs. The first, Listing 12-1, creates the vocabulary file that is read by the main program and needs to be run only once. This vocabulary-creating program is self-explanatory and consists mainly of a list of words. The second program, Listing 12-2, is the actual Word Square program. Its variables, arrays, and subroutines are documented on the following pages. Figure 12-1 shows the flowchart for the Word Square program.

WORD SQUARE VARIABLES

The following variables are used in Word Square:

LD—Load flag, which prevents reloading of files.

MV—Maximum value of a word.

MW\$—Word that corresponds to MV.

WD\$—Temporary storage for words.

WL—Character length of words.

I—Loop counter.

NT—Current turn number.

TT—Total number of turns in the game.

DU—Turn duration in minutes.

FQ—Frequency value for musical notes.

NP—Number index for the current player.

TP—Total number of players.

SR—Letter score, which is equal to the sum of all letters that form valid words in the square.

NW—Number of valid words formed in the word square.

IO\$—Variable used to hold keyboard input.

V—Used to compute the sum of letter values for a validly formed word.

D—Delay loop counter.

BW—Loop counter (1-5) used to denote the word in the box that is being input or evaluated.

E—Error flag. If E=1, then an error has been detected.

T—Temporary storage, which is most often used to hold the ASCII value of an input character.

PI\$—Temporary storage for word strings.

LT\$—Contains the seven random letters players

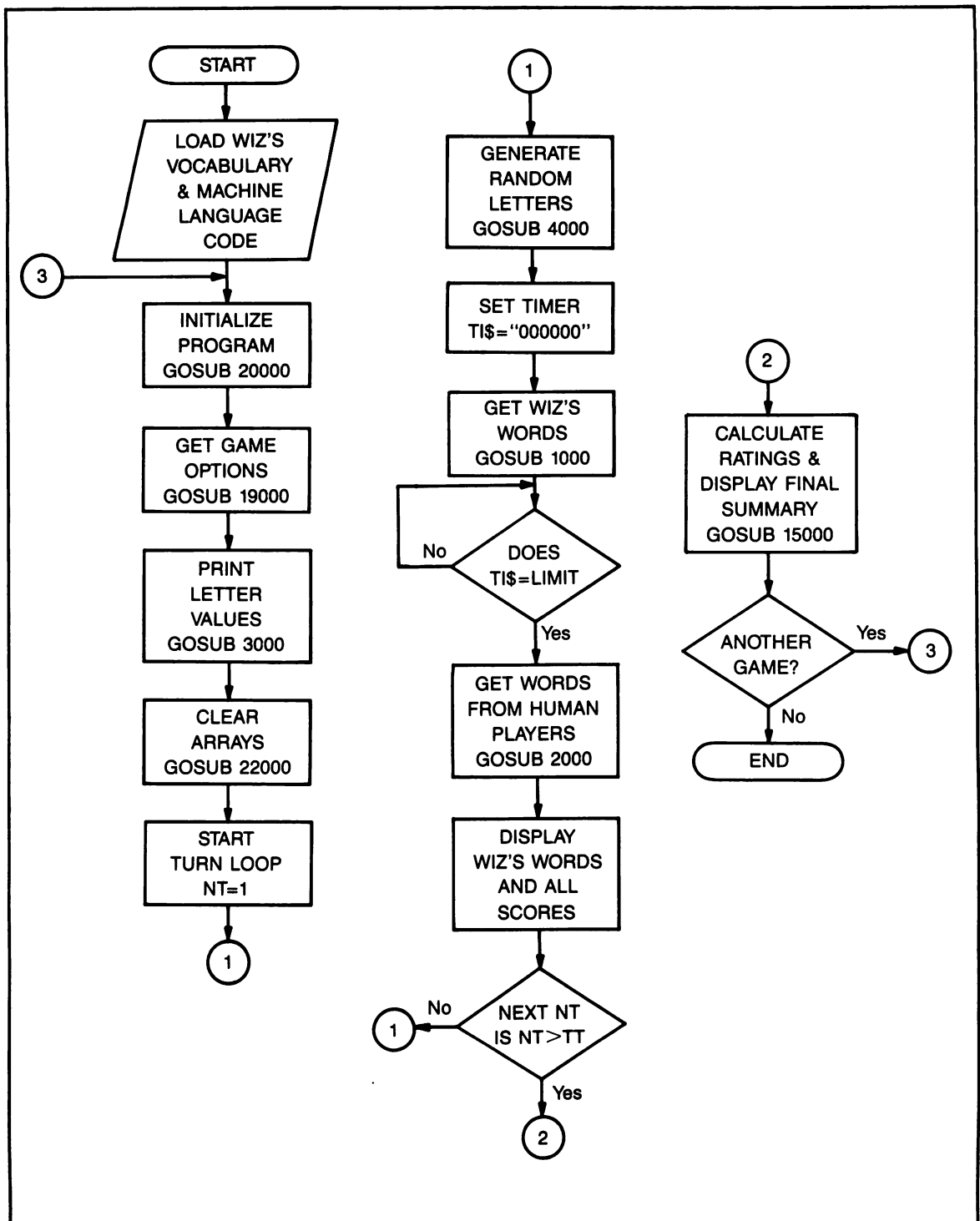


Fig. 12-1. The flowchart for the Word Squares program.

Listing 12-1. Vocabulary program.

```

100 REM THIS PROGRAM WRITES THE VOCABULARY FILE
120 OPEN 5,8,5,"O:VOCABL.DAT,P,W"
122 REM WRITE LOAD ADDRESS
125 PRINT#5,CHR$(0);CHR$(64);
130 READ W$:L=LEN(W$)
140 PRINT#5,LEFT$(W$,L-1);
150 PRINT#5,CHR$(ASC(RIGHT$(W$,1))+128);
160 IF W$<>"ZZ" THEN 130
170 CLOSE 5
180 END

31031 DATA AB,AD,AN,NA,FA,AM,AR,AS
31032 DATA AT,TA,AW,AX,AY,YA,BE,BO
31033 DATA BY,DE,ID,DO,OD,EF,EH,HE
31034 DATA EL,EM,ME,EN,OE,PE,ER,RE
31035 DATA ET,WE,EX,YE,IF,OF,GO,HI
31036 DATA HO,OH,SH,LI,MI,IN,PI,IS
31037 DATA SI,IT,PI,IS,SI,IT,TI,XI
31038 DATA JO,LO,OM,MU,UM,MY,NO,ON
31039 DATA NU,OR,OS,SO,TO,OW,OX,OY
31040 DATA UP,US,UT,XU,ABA,BAA,AGA,AHA
31041 DATA ALA,AMA,ANA,CAB,BAD,DAB,BAG,GAB
31042 DATA BAH,JAB,KAB,ALB,BAL,LAB,BAN,NAB
31043 DATA ABO,BOA,BAR,BRA,BAT,TAB,ABY,BAY
31044 DATA CAD,ACE,LAC,CAM,MAC,CAN,OCA,CAPADY.
31045 DATA ARC,CAR,SAC,ACT,CAT,CAW,CAY,ADD
31046 DATA DAD,FAD,DAG,GAD,DAH,HAD,AID,DAK
31047 DATA LAD,DAM,MAD,AND,ADO,DAP,PAD,RAD
31048 DATA SAD,TAD,WAD,DAY,ADZ,AGE,HAE,KEA
31049 DATA ALE,LEA,MAE,NAE,APE,PEA,ARE,EAR
31050 DATA ERA,SEA,ATE,EAT,ETA,TAE,TEA
31051 DATA AWE,AYE,YEA,FAG,FAN,DAF,ARF,FAR
31052 DATA AFT,FAT,FAX,FAY,GAG,HAG,JAG,GAL
31053 DATA LAG,GAM,MAG,GAN,NAG,AGO,GOA,GAP
31054 DATA GAR,RAG,GAS,SAG,GAT,TAG,WAG,GAY
31055 DATA HAH,HAM,HAP,FAH,RAH,ASH,HAS,HAT
31056 DATA HAW,HAY,YAH,AIL,AIM,ANI,AIR
31057 DATA JAM,JAR,RAJ,TAJ,JAW,JAY,ALK,KOA
31058 DATA OAK,OKA,ARK,ASK,KAT,AUK,KAY,YAK
31059 DATA ALL,LAM,ALP,LAP,PAL,LAR,SAL
31060 DATA LAT,ULA,AWL,LAW,LAY,MAM,MAN,MOA
31061 DATA AMP,MAP,ARM,MAR,RAM,MAT,TAM,MAW
31062 DATA MAY,YAM,NAP,PAN,RAN,NAR,NT,TAN
31063 DATA VAN,AWN,WAN,ANY,NAY,OAR,OAT,OVA
31064 DATA ZOA,PAP,PAR,RAP,ASP,PAS,SAP,SPA

```

31065 DATA APT, PAT, TAP, PAW, WAP, PAX, PAY, PYA
 31066 DATA YAP, ZAP, QUA, ART, RAT, TAR, RAW, WAR
 31067 DATA RAY, RYA, YAR, ASS, SAT, SAW, WAS, SAX
 31068 DATA SAY, TAT, TAU, UTA, TAV, VAT, TAW, WAS
 31069 DATA TAX, WAW, WAX, WAY, YAW, ZAX, EBB, BIB
 31070 DATA BOB, BUB, COB, CUB, BED, BID, BUD, DUB
 31071 DATA BEE, BEG, BEL, BEN, NEB, BET, WEB
 31072 DATA BYE, FIB, FOB, FUB, BIG, GIB, BOG, GAB
 31073 DATA BUG, HOB, HUB, JIB, LIB, MIB, BIN, BIB
 31074 DATA BIO, RIB, BIS, SIB, BAT, JOB, LOB
 31075 DATA BUM, BON, NOB, BUN, NUB, BOO, BOP
 31076 DATA ROB, SOB, BOT, BOW, BOX, BOY, FUB, RUB
 31077 DATA BUS, SUB, BUT, TUB, BUY, COD, DOC, COG, TOD
 31078 DATA CEE, ICE, CEP, CUE, ECU, COG, CHI
 31079 DATA PIC, SIC, TIC, ICY, COL, CUM, CON, COO
 31080 DATA COP, ROC, COS, COW, COX, COY, COZ
 31081 DATA CUP, CUR, CRY, CUT, DID, ODD, DUD, FED
 31082 DATA DEG, DIE, ELD, LED, DEN, END, DOE
 31083 DATA RED, TED, DUE, DEV, DEW, WED, DEX
 31084 DATA DYE, ZED, FID, DIG, GID, DOG, GOD, DUG
 31085 DATA HID, HOD, KID, LID, DIM, MID, DIN, DIP
 31086 DATA RID, DIT, JOD, DOL, OLD, DOM, MOD
 31087 DATA DON, NOD, DUN, FOD, DOR, ROD, SOD, DOT
 31088 DATA TOD, DUO, UDO, DOW, YOD, URD, DRY, FEE
 31089 DATA GEE, JEE, EKE, EEL, LEE, NEE, PEE
 31090 DATA REE, SEE, TEE, EVE, VEE, EWE, WEE, EYE
 31091 DATA ZEE, EFF, FIE, KEF, ELF, FEN, NEF, FOE
 31092 DATA FET, FEU, FEW, FEY, FEZ, EGG, KEG
 31093 DATA LEG, GEM, ENG, EGO, PEG, ERG, GET, TEG
 31094 DATA GEY, HIE, HEM, HEN, HOE, HEP, HER, SEE
 31095 DATA ETH, THE, HUE, HEW, HEX, HEY, LEI, LID
 31096 DATA PIE, IRE, TIE, VIE, JOE, JET, ELK, LEK
 31097 DATA KEN, OKE, KEP, UKE, KEX, KEY, ELL, ELM
 31098 DATA LET, LEU, LEV, LYE, MEN, REM, MESTEP
 31099 DATA EMU, MEW, EON, ONE, PEN, SEN, NET, TEN
 31100 DATA NEW, WEN, YEN, OPE, ORE, ROE, TOE
 31101 DATA OWE, WOE, PEP, PET, PEW, YEP, ERR, ERS
 31102 DATA RES, RET, RUE, REV, REX, RYE, ESS, SAT
 31103 DATA SUE, USE, SEW, SEX, YES, VET, TEW, WET
 31104 DATA VEX, WYE, YEW, OFF, FIG, FOG
 31105 DATA FUG, FOH, KIF, FIL, FIN, FIR, FIT, FIX
 31106 DATA FLU, FLY, FUN, FOP, FOR, FRO, OFT, FOU
 31107 DATA FOX, FOY, FUR, FRY, GIG, HOG, HUG, UGH
 31108 DATA JIG, MIG, GIN, GIP, PIG, RIG, WIG, ZIG
 31109 DATA JOG, JUG, LOG, LUG, GUM, MUG, GYM, NOG
 31110 DATA GNU, WHO, ORUN, PHT, HUP, SHY, HUT, THY

31111 DATA WHY, JIN, ILK, INK, KIN, KIP, IRK, SKI
 31112 DATA KIT, ILL, MIL, LIN, NIL, OIL, LIP, TIL
 31113 DATA MIM, NIM, IMP, MIR, RIM, ISM, SIM, VIM
 31115 DATA TIN, WIN, NIX, POI, PIP, RIP, SIP, PIT
 31116 DATA TIP, YIP, ZIP, SIR, SIS, IST, ITS
 31117 DATA SIT, VIS, WIS, SIX, TIT, TUI, WIT
 31118 DATA JOT, JOW, JOY, JUN, JUS, JUT, KOP
 31119 DATA KOS, WOK, TSK, SKY, LOO, LOP, SOL, LOT
 31120 DATA LOW, OWL, LOX, PUL, PLY, SLY, LUX, MOM
 31121 DATA MUM, MON, MOO, MOP, MOT, TOM, MOW, UMP
 31122 DATA RUM, NUN, NOR, SON, NOT, TON, VOW
 31123 DATA NOW, OWN, WON, YON, PUN, RUN, URN, SUN
 31124 DATA NUT, TUN, TOO, WOO, WOO, ZOO, POP
 31125 DATA SOP, POT, TOP, ORT, ROT, TOR, OUR, ROW
 31126 DATA SOT, SOU, SOW, SOX, SOY, TOT, OUT, TWO
 31127 DATA TOW, TOY, YOU, VOW, WOW, YOW, OXY
 31128 DATA PUS, SUP, SPY, PUT, TUP, PYX, RUST
 31129 DATA TRY, STY, TUT, TUX, AJAR, HADJ, HAJI, HAJJ
 31130 DATA JACK, JADE, JADY, JAIL, JAMB, JAFE, JARL, JAZZ
 31131 DATA JEAN, JEEP, JEER, JERK, JESS, JEST, JIBE, JILL
 31132 DATA JILT, JINN, JINX, JIVE, JOAD, JOIN, JOKE, JOLE
 31133 DATA JOLT, JOWL, JBA, JUBE, JUJU, JUMP, JUNK, JURA
 31134 DATA JURY, JUST, JUTE, RAJA, SOJA, BIJOU, EJECT
 31135 DATA ENJOY, FJORD, HAJJI, JABOT, JACAL, JACKY, JAGGY, JAKES
 31136 DATA JALAP, JAMBE, JAUNT, JAZZY, JEBEL, JEHAD, JEMMY, JENNY
 31137 DATA JERKY, JERRY, JETTY, JEWEL, JIHAD, JIMMY, JINNI, JACKO
 31138 DATA JOINT, JOIST, JOLLY, JORAM, JORUM, JOULE, JOUST, JUDGE
 31139 DATA JUGAL, JUICE, JUICY, JULEP, JUMBO, JUMPY, JUMPY, JUNCO
 31140 DATA JUNTA, JUPON, JURAL, JURAT, JUROR, JUTTY, MAJOR, MUJIK
 31141 DATA RAJAH, SAJOU, SLOJD, THUJA, QUAD, QUAG, QUAY, QUID
 31142 DATA QUIP, QUIT, QUIZ, QUOD, EQUAL, EQUITY, FIQUE, PIQUE
 31143 DATA QUACK, QUAFF, QUAIL, QUAKE, QUALM, QUART, QUASH, QUEAN
 31144 DATA QUEEN, QUEER, QUELL, QUERY, QUEST, QUEUE, QUICK, QUIET
 31145 DATA QUILL, QUILT, QUINT, QUIPU, QUIRE, QUIRK, QUIRT, QUITE
 31146 DATA QUOIN, QUOIT, QUOTA, QUOTE, SQUAB, SQUAD
 31147 DATA SQUAT, SQUAW, SQUIB, SQUID, TOQUE, TUQUE, USQUE, APEX
 31148 DATA AXIL, AXIS, AXLE, AXON, COAX, COXA, CRUX, EAUX
 31149 DATA EXIT, FLAX, FLEX, FLUX, FOXY, HOAX, IBEX, ILEX
 31150 DATA IXIA, JINX, LYNX, MINX, MOXA, NEXT, ONYX, ORYX
 31151 DATA OXEN, OXID, OXIM, PIXY, KEYS, TAXI, TEXT, XYST
 31152 DATA ADDAX, ADMIX, AFFIX, ANNEX, ATAXY, AXIAL, AXILE, AXIOM
 31153 DATA AXLED, AXMAN, AXONE, BEAUX, BORAX, BOXER, BRAXY, BUXOM
 31154 DATA CALYX, CIMEX, CODEX, CYLIX, EXACT, EXALT, EXCEL, EXERT
 31155 DATA EXILE, EXIST, EXPEL, EXTOL, EXTRA, EXUDE, EXULT
 31156 DATA GALAX, HELIX, HEXAD, HEXYL, HYRAX, INDEX, INFIX, IXTLE
 31157 DATA KYLIX, LATEX, LAXLY, MAXIM, MIXER, MUREX, NIXIE, OXBOW

31158 DATA OXEYE, OXIDE, OXIME, OXLIP, PHLOX, PROXY, PYXIE, PYXIS
 31159 DATA RADIX, RELAX, REMEX, SILEX, SIXTH, SIXTY, TOXIC, TOXIN
 31160 DATA UNIX, UNSEX, VARIX, VIXEN, WAXEN, XENIA, XENON
 31161 DATA XERUS, XYLAN, XYLEM, XYLOL, XYLYL, ADZE, BUZZ, COZ
 31162 DATA COZY, CZAR, DAZE, DOZE, DOZY, FAZE, FEZ, FIZZ
 31163 DATA FRIZ, FUZE, FUZZ, GAZE, HAZE, HAZY, JAZZ, LAZE
 31164 DATA LAZY, MAZE, MAZY, OOZE, OOZY, OYEZ, QUIZ, RAZE
 31165 DATA SIZE, TZAR, WHIZ, ZANY, ZAX, ZEAL, ZEBU, ZEIN
 31166 DATA ZERO, ZEST, ZETA, ZINC, ZIP, ZOA, ZOIC, ZONE
 31167 DATA ZOOM, ZOON, ZYME, AMAZE, AZOIC, AZOTE, AZURE, BAZAR
 31168 DATA BEZEL, BLAZE, BONZE, BOOZE, BORTZ, BRAZA, BRAZE, COLZA
 31169 DATA COZEN, CRAZE, CRAZY, CROZE, DIZEN, DOZEN, ENZYM, FIZZY
 31170 DATA FRIZZ, FURZY, FUZEE, FUZIL, FUZZY, GAUZE, GAUZY, GLAZE
 31171 DATA GRAZE, HAZEL, HUZZA, JAZZY, KAZOO, LAZAR, MAIZE, MAZER
 31172 DATA MEZZO, MIRZA, MIZEN, OUZEL, OZONE, PLAZA, PRIZE, RAZEE
 31173 DATA RAZOR, SEIZE, SEZIN, SIZAR, SIZED, SIZER, SOZIN, SPITZ
 31174 DATA TOPAZ, VIZIR, VIZOR, WALTZ, WINZE, WIZEN, ZAMIA, ZAYIN
 31175 DATA ZEBEC, ZEBRA, ZIBET, ZINKY, ZLOTY, ZOMBI, ZONAL, ZOOID
 31176 DATA ZORIL, SPIT, SPIV, WISP, PITY, YIRR, STIR, WRIT
 31177 DATA WIRY, SUIT, TWIT, KOOK, LOOK, YOLK, LURK, SULK
 31178 DATA MONK, MURK, MUSK, NOOK, KNOP, KNOT, KNOW, PUNK
 31179 DATA KNUR, SUNK, ROOK, KOTO, TOOK, PORK, POKY, WORK
 31180 DATA RUSK, TUSK, LOLL, LULL, MOLL, MULL, NULL, POLL
 31181 DATA ROLL, TOLL, PULL, LULU, LOOM, MOOL, MOLT, MOLY
 31182 DATA LUMP, PLUM, SLUM, LOON, LORN, ONLY, NURL, LUNT
 31183 DATA LUNY, LOOP, POLO, POOL, SOLO, LOST, TOOL, WOOL
 31184 DATA PLOP, SLOP, PLOT, PLOW, PLOY, ROTL, LORY, LOSS
 31185 DATA LOST, SLOT, SOUL, SLOW, LOUT, TOLU, VOLT, YOWL
 31186 DATA PULP, PURL, PLUS, SLUR, MONO, MOON, MORN, NORM
 31187 DATA MONS, MUON, MOWN, MOOR, ROOM, MOOT, FOMP, FROM
 31188 DATA ROMP, MORT, WORM, MOSS, MOST, PUMP
 31189 DATA PUMP, RUMP, SUMP, TUMP, MUSS, MUST, SMUT, STUM
 31190 DATA SWUM, MUTT, NOON, NOUN, SUNN, POON, SOON, ONTO
 31191 DATA TOON, WOON, PORN, PONS, UPON, FONY, SORN, TORN
 31192 DATA WORN, SNOT, NOUS, SNOW, SOWN, NOSY, UNTO, NOWT
 31193 DATA TOWN, WONT, TONY, SPUN, PUNT, FUNY, RUNT, TURN
 31194 DATA STUN, POOP, POOR, ROOP, OOPS, ROOT, ROT, SOOT
 31195 DATA TOOT, PROP, PORT, POUR, PROW, ROPY, POST, SPOT
 31196 DATA STOP, OPUS, SOUP, POSY, POUT, TYPO, TORR, SORT
 31197 DATA SOUR, ROSY, TORT, TROT, ROUT, TOUR, TROW, RYOT
 31198 DATA TROY, TYRO, ROUX, YOUR, SOSS, POST, OUST
 31199 DATA SLOW, TOUT, TYPP, PURR, SPUR, SPRY, PUSS, PUTT
 31200 DATA RUST, URUS, YURT, TUTU, GLOP, SLOG, FLOW, LOGY
 31201 DATA GULP, PLUG, SLUG, GLUT, UGLY, SMOG, GRUM, SMUG
 31202 DATA GOON, SONG, TONG, GOWN, PUNG, RUNG, SNUG, SUNG
 31203 DATA FROG, FOGY, GROT, GROW, GORY, GOUT

31204 DATA GURU, GUST, HUSH, KISH, KITH, HILL, HILT, SHIM
 31205 DATA WHIM, HISH, SHIN, SINH, HINT, THIN, THIO, PISH
 31206 DATA SHIP, PITH, WHIP, WHIR, HISS, HIST, SHIT, SITH
 31207 DATA THIS, SHIV, WISH, WHIT, WHIZ, WITH, KOHL, HULK
 31208 DATA HONK, HUNK, HOOK, KOPHUL, HOLM
 31209 DATA HOLP, HOLT, LOTH, HOWL, HOLY, HURL, LUSH, HYMN
 31210 DATA MOHR, MOTH, WHOM, HOMO, HUMP, MUSH, MYTH, HORN
 31211 DATA NOSH, SHUN, HUNT, HOOP, POOH, SHOO, HOOT, QOPH
 31212 DATA POSH, SHOP, SOPH, PHOT, WHOP, HYPO, THRO, HOUR
 31213 DATA HOST, SHOT, TOSH, SHOW, THOU, PUSH, RUSH, HURT
 31214 DATA RUTH, THRU, HUSS, SHUT, THUS, TUSH, KIWI, MINI
 31215 DATA IMPI, NISI, TIPI, IRIS, IWIS, TITI, KINK, KIRK
 31216 DATA KILL, MILK, LIMP, SLIM, MILT, LIMY, LINN, LION,
 LOIN, NOIL
 31217 DATA LINT, INLY, LINY, OLIO, ROIL, SILO, SOIL, TOIL
 31218 DATA VIOL, OILY, LISP, SLIP, PULI, PILL, TIRL, LIST
 31219 DATA SLIP, PULI, PILY, TIRL, LIST, SLIT, SILT, TILT
 31220 DATA WILT, WILY, MINT, OMIT, PIMP, PRIM, SIMP, WIMP
 31221 DATA TRIM, MIRY, RIMY, MISS, MIST, SWIM, MITT, IRON
 31222 DATA INTO, WIND, YONI, PRIN, SNIP, SPIN, FINT, FINY
 31223 DATA RUIN, TINT, UNIT, TWIN, TINY, VINY, WINY, TOPI
 31224 DATA SORI, RIOT, TIRO, TORI, TRIO, ZORI, PIPY, TRIP
 31225 DATA MORE, OMER, SOME, MOTE, TOME, MOVE, MEOW, TERM
 31226 DATA MESS, STEM, MUSE, SMEW, MUTE, NEON, NONE, NOPE
 31227 DATA OPEN, PEON, PONE, NOSE, NOTE, TONE, OVEN, ENOW
 31228 DATA PENT, RENT, TERN, RUNE, WREN, NESY, TEST, SENT
 31229 DATA SEWN, TENT, TUNE, VENT, WENT, NEWT, TYNE, ENVY
 31230 DATA PEPO, POPE, PORE, ROPE, EPOS, PESO, POSE, POET
 31231 DATA TOPE, EXPO, ROSE, SORE, ROTE, TORE, ROUE, OVER
 31232 DATA WORE, OYER, YORE, OYES, TOTE, VETO, WOVE, OYEZ
 31233 DATA PREP, REPP, PURE, PREY, PYRE, STEP
 31234 DATA SPUE, SPEW, ESPY, WEPT, TYPE, RUER, ERST, REST
 31235 DATA RUSE, SURE, USER, TRET, TRUE, VERT, TREY, VERY
 31236 DATA STET, TEST, SUET, VEST, STEW, WEST, GUFF, HUFF
 31237 DATA MIFF, RIFF, TIFF, IFFY, LUFF, MUFF, TOFF, PUFF
 31238 DATA RUFF, TUFF, FRIG, GIFT, FLOG, GOOF
 31239 DATA FROG, FOGY, FRUG, FISH, HOOF, FINK, FILL, FILM
 31240 DATA FOIL, FLIP, FLIT, LIFT, FLIX, FIRM, FIRN, FRIT
 31241 DATA RIFT, FIST, SIFT, FOLK, FUNK, FORK, FULL, FOOL
 31242 DATA FLOP, LOFT, FOUL, FLOW, FOWL, WOLF, FURL, FLUX
 31243 DATA FORM, FROM, FUMY, FONT, ROOF, POOF, PROF
 31244 DATA POUF, FORT, FOUR, FROW, FOSS, SOFT, TOFT, FOXY
 31245 DATA FOZY, SURF, TURF, FURY, FUSS, TUFT, MIGG, GRIG
 31246 DATA GONG, NOGG, GOGO, GROG, HIGH, NIGH, SIGH, HONG
 31247 DATA HUNG, GOSH, SHOG, PUGH, GUSH, THUG, GINK, KING
 31248 DATA GILL, GLIM, LING, GIRL, GILT, LIMP, GRIM

31249 DATA PING, GRIN, RING, SIGN, SING, TING, WING, ZING
 31250 DATA GIRO, YOGI, GRIP, PRIG, GIRT, GRIT, TRIG, GYRI
 31251 DATA GIST, SWIG, TWIG, GUNK, GOOK, GOWK, GULL, GLUM
 31252 DATA LONG, LUNG, LOGO, HUGE, EGIS, GIVE, SKEG, GLEN
 31253 DATA LOGE, OGLE, GELT, GERM, GEUM, GOES, GENS, GENT
 31254 DATA ERGO, GOER, GORE, GOES, SEGO, URGE, GREW, GREY
 31255 DATA GYRE, GEST, GYVE, HETH, HIKE, HEIR, HIRE, HELL
 31256 DATA HELM, HOLE, HELP, HERL, HOME, HEMP, HERM, HENT
 31257 DATA THEN, HEWN, WHEN, HOPE, HERO, HOER, HOSE, SHOE
 31258 DATA HOVE, PHEW, HYPE, HERS, RESH, HEST, SHEW, TETH
 31259 DATA THEW, WHET, HYTE, THEY, WHEW, WHEY, JIVE, LIKE
 31260 DATA MIKE, KINE, KEPI, PIKE, KEIR, TIKE, LIME, MILE
 31261 DATA LIEN, LINE, PILE, PLIE, RIEL, RILE, ISLE, TILE
 31262 DATA LIEU, EVIL, LIVE, VEIL, VILE, ILEX, MIME, MIEN
 31263 DATA MINE, ETUI, WITE, YETI, VIE, WIVE, JUKE, JUPE
 31264 DATA KELL, KELP, YELK, MOKE, KEMP, MERK, KEND, KERN
 31265 DATA NUKE, KNEW, POKE, SOKE, EMIR, MIRE, RIME, MISE
 31266 DATA EMIT, ITEM, MITE, TIME, NINE, FINE, REIN, SINE
 31267 DATA TINE, VEIN, VINE, WINE, ZEIN, PIPE, PERI, PIER
 31268 DATA RIPE, SIPE, WIPE, YIPE, REIS, RISE, SIRE, RITE
 31269 DATA TIER, RIVE, WEIR, WIRE, SITE, VISE, WISE, SIZE
 31270 DATA ETUI, WITE, EXIT, YETI, VIEW, WIVE, JOKE, JOES
 31271 DATA JOEY, JUPE, JESS, KELL, KELP, YELK, MOKE, KEMP
 31272 DATA MERK, KEND, KERN, KENT, NUKE, KNEW, POKE, SOKE
 31273 DATA KETO, TOKE, WOKE, YOKE, PERK, PUKE
 31274 DATA TREK, YERK, SKEW, KYTE, TYKE, MELL, SELL, TELL
 31275 DATA WELL, YELL, MOLE, MERL, MELT, MULE, MEWL, ELMY
 31276 DATA ENOL, LENO, LONE, LENS, LENT, LUNE, OLEO, POLE
 31277 DATA LORE, ORLE, ROLE, LOSE, SLOE, SOLE, LEVO, LOVE
 31278 DATA VOLE, PELT, PYLE, YELP, LURE, RULE, LYRE, RELY
 31279 DATA LESS, LEST, LUES, SLUE, SLEW, LYSE, LUTE, TULE
 31280 DATA WELT, LUXE, YULE, LEVY, MEMO, NOME, OMEN, MENU
 31281 DATA MOPE, POEM, POME, DYNE, DOPE, OPED, DOER, REDO,
 RODE, DOES, DOSE, DOTE, TOED
 31282 DATA SPED, DUPE, DURE, RUDE, RUED, DREW, DYER, SUED
 31283 DATA USED, DUET, DEWY, DOFF, DUFF, FIND, FIDO, FOLD
 31284 DATA FOND, FUND, FOOD, FORD, GILD, DING, GIRD, GRID
 31285 DATA GOLD, DONG, DUNG, GOOD, DOGY, DRUG, HIND, HOLD
 31286 DATA HOOD, SHOD, DHOW, THUD, MIDI, NIDI, DINK, KIND
 31287 DATA DIRK, DISK, SKID, DILL, MILD, IDOL, DIRL, SLID
 31288 DATA WILD, IDLY, IDYL, MIND, DONI, NODI, RIND, DINT
 31289 DATA WIND, DOIT, VOID, DRIP, QUID, DIRT, TIDY, DOJO
 31290 DATA JUDO, DUNK, DUSK, KUDU, DOLL, DULL, MOLD, PLOD
 31291 DATA LORD, SOLD, DOLT, TOLD, LOUD, DOWL, WOLD, DULY
 31292 DATA DOOM, MOOD, DORM, DUMP, DRUM, POND, UNDO, DOWN
 31293 DATA YOND, RYND, WYND, DOOR, ODOR, ORDO, ROOD, WOOD

31294 DATA DORP, DROP, PROD, QUOD, TROD, DOUR, DURO, WORD
 31295 DATA DORY, DOSS, TODAY, DOXY, DOZY, SPUD, SURD, TURD
 31296 DATA SUDS, DUST, STUD, DUTY, EPEE, KEEF, FEEL, FLEE
 31297 DATA FEME, FERE, FREE, REEF, FEET, FETE, GHEE, GEEK
 31298 DATA GLEE, GENE, OGEE, GREE, HEEL, HEME, HERE, THEE
 31299 DATA JEEZ, KEEK, KEEL, LEEK, MEEK, KEEN, KNEE, KEEP
 31300 DATA FEEK, REEK, SEEK, SKEE, WEEK, PEEL, LEER, REEL
 31302 DATA MEET, METE, TEEM, EMEU, NEEP, PEEN, ERNE, ESNE
 31303 DATA SEEN, TEEN, EVEN, NEVE, PEEP, PEER, SEEP, VEEP
 31304 DATA WEEP, SEER, SERE, RETE, TREE, EVER, VEER, EWER
 31305 DATA WERE, EERY, EYRE, FIEF, FIFE, HEFT, FILE
 31306 DATA LIEF, LIFE, FINE, FIRE, RIFE, FIVE, WIFE, KERF
 31307 DATA FYKE, FELL, FLOE, PELF, SELF, FELT, LEFT, FLUE
 31308 DATA FUEL, FLEW, FLEX, FUME, FERN, FORE, NOTROE, SERF
 31309 DATA FRET, FESS, FUSE, WEFT, FUZE, YEGG, EPIC, PICE
 31310 DATA ERIC, RICE, SICE, CITE, VICE, KECK, NECK, COKE
 31311 DATA PECK, RECK, CUKE, CELL, COLE, CELT, CLUE, LUCE
 31312 DATA CLEW, COME, CYME, CONE, ONCE, CENT, COPE, CORE,
 COTE, COVE, PUCE, CURE, ECRU
 31314 DATA SECT, SYCE, CUTE, CUFF, COIF, FICO, FISC, FLOC
 31315 DATA COOF, CORF, CURF, CHUG, GUCK, CLOG, HICK, CHIN
 31316 DATA INCH, CHIP, RICH, CHIT, ITCH, HUIC, HOCK, LOCH
 31317 DATA CHUM, MUCH, CHOP, COSH, DUCH, SUCH, KICK, LICK
 31318 DATA NICK, PICK, RICK, SICK, TICK, WICK, ICKY, COIL
 31319 DATA LOCI, CLIP, CION, COIN, ICON, UNCI, COIR, OTIC
 31320 DATA URIC, CIST, CITY, JOCK, LOCK, LUCK, MOCK, MUCK
 31322 DATA RUCK, CUSK, SUCK, TUCK, CULL, CULM, COOL, LOCO
 31323 DATA CLOT, COLT, COWL, CLOY, CURL, CULT, COMP, CORM
 31324 DATA SCUM, COON, CORN, UNCO, CONY, SYNC, COOP, POCO
 31325 DATA COOT, CROP, SCOP, COUP, COPY, CROW, COST, SCOT
 31326 DATA SCOW, COSY, CUSP, SCUP, CURR, CRUS, CURT, CUSS
 31327 DATA SCUT, CYST, DEED, DIED, DUDE, DYED, EDDY, DIDO
 31328 DATA DONOT, RUDD, SUDD, FEED, EDGE, HEED, DELE
 31329 DATA DEEM, DEME, MEED, DENE, NEED, DEEP, PEED, DEER
 31330 DATA DREE, REED, SEED, WEED, EYED, DELF, FLED, FEND
 31331 DATA DEFT, FEUD, DEFY, GELD, DOGE, EDGY, HIDE, HELD
 31332 DATA HOED, HERD, SHED, HUED, DIKE, DEIL, DELI, IDLE
 31333 DATA LID, DIME, IDEM, DINE, NIDE, PIED, RIDE
 31334 DATA IDES, SIDE, DIET, EDIT, TIDE, TIED, DIVE, VIED
 31335 DATA WIDE, DESK, DUKE, DYKE, DELL, MELD, LEND, DOLE
 31336 DATA LODE, SLED, DUEL, VELD, LEWD, WELD, MEND, DEMO
 31337 DATA DOME, MODE, DEMY, DONE, NODE, PEND, REND, SEND
 31338 DATA DET, TEND, DUNE, NUDE, UNDE, VEND, WEND, DENY
 31339 DATA STAY, SWAY, TAUT, WATT, WAVY, WAXY, BIBB, BLEB
 31340 DATA BLOB, BULB, BOMB, BOOB, BUBO, BICE, BECK, CUBE
 31341 DATA CHUB, CRIB, BUCK, BLOC, CLUB, COMB, CURB, BIDE

31342 DATA BLED, BEND, BODE, BRED, DEBT, BIND, BIRD, DRIE
 31343 DATA DIS, BOLD, DUMB, BOND, BODY, DRUB, BEEN
 31344 DATA BEEP, BEER, BREE, BEET, GIBE, BERG, GYBE, HERB
 31345 DATA BETH, JIBE, BIKE, KIBE, BILE, BINE, BIER, BISE
 31346 DATA BITE, BIZE, JUBE, KERB, BELL, BOLE, LOBE, PLEB
 31347 DATA BELT, BLUE, LUBE, BLEW, BERM, BONE, EBON, BENT
 31348 DATA OBE, BORE, ROBE, OBEX, OBEY, RUBE, BREW
 31349 DATA BYRE, BEST, BYES, TUBE, BYTE, BEVY, BIFF, BUFF
 31350 DATA FLUB, FORB, GLIB, BING, BRIG, GLOB, BUNG, BOGY
 31351 DATA GOBY, BURG, GRUB, BUHL, HOBO, BOSH, BOTH, BUSH
 31352 DATA IBIS, BILK, BISK, BILL, LIMB, BOIL, BLIP, BIRL
 31353 DATA BRM, NIBS, INBY, OBIT, BIRR, BRIT, BULK
 31354 DATA KNOB, BUNK, BOOK, BOSK, BUSK, BOLL, BULL, BOLO
 31355 DATA LOBO, OBOL, SLOB, BLOT, BOLT, BLOW, BOWL, BLUR
 31356 DATA BURL, SLUB, NUMB, BOOM, TOMB, UMBO, WOMB, BUMP
 31357 DATA BOON, BORN, SNOB, BONY, BURN, SNUB, BUNT, BOOR
 31358 DATA BRO, BOOT, BOZO, SORB, BORT, BROW, BOSS, STOB
 31359 DATA BOTT, BOUT, TOBY, BOUY, BOXY, BURP, BURR, BRUT
 31360 DATA BURY, RUBY, BUSS, BUST, STUB, BUSY, BUTT, CHIC
 31361 DATA COCK, COCO, CEDE, DICE, ICED, DECK, CODE, COED
 31362 DATA CUED, DUCE, DICK, ODIC, DISC, DOCK, DUCK, CLOD
 31363 DATA COD, CORD, CRUD, CURD, SCUD, DUCT, CERE
 31364 DATA CETE, CHEF, FICE, FECK, CLEF, GECK, HECK, LECH
 31365 DATA ECHO, ETCH, CHEW, CHEZ, CEIL, LICE, MICE, NICE
 31366 DATA FAIL, ARIL, LAIR, LIAR, LIRA, RAIL, RIAL, SAIL
 31367 DATA ALIT, LATI, TAIL, TALI, VAIL, VIAL, WAIL, AXIL
 31368 DATA MAIM, MAIN, MINA, AMIR, RAMI, MAXI, NIFA
 31369 DATA PAIN, FIAN, RAIN, RANI, SAIN, ANTI, TAIN, VAIN
 31370 DATA VINA, WAIN, AYIN, IOTA, PAIR, SARI, VAIR, AIRY
 31371 DATA IZAR, VISA, WAIT, VIVA, JAUJ, JARL, JATO, JOTA
 31372 DATA JAUP, LANK, KOLA, LARK, TALK, WALK, LAKY, AMOK
 31373 DATA MARK, MASK, KAON, KOAN, KNAP, KNAR, NARK, RANK
 31374 DATA SANK, TANK, YANK, OKRA, SOAK, KAYO, OKAY, PARK
 31375 DATA SARK, KART, SKAT, TASK, SKUA, KYAT, LALL, MALL
 31376 DATA OLLA, FALL, TALL, WALL, ALLY, MALM, LOAM, MOLA
 31377 DATA LAMP, PALM, MARL, ALMS, SLAM, MALT, ALUM, MAUL
 31378 DATA AMYL, LOAN, PLAN, LUNA, ULNA, LAWN, OPAL, ORAL
 31379 DATA ALSO, ALTO, LOTA, TOLA, OVAL, ALOW, PALP, SLAP
 31380 DATA FLAT, PAWL, Paly, PLAY, LASS, LAST, SALT, SLAT
 31381 DATA SLAW, SLAY, LUAU, WALY, YAWL, AMMO, MOAN, NOMA
 31382 DATA MAUN, MANY, MYNA, MORA, ROAM, SOMA, ATOM, MOAT
 31383 DATA MOXA, PRAM, RAMP, SAMP, TAMP, PUMA, VAMP, MART
 31384 DATA TRAM, MURA, WARM, ARMY, MASS, MAST, SWAM, MAZY
 31385 DATA ANON, ROAN, NONA, NAOS, NOTA, NOVA, AXON, SNAP
 31386 DATA SPAN, PANT, PAWN, RANT, WARN, NARY, YARN, SANS
 31387 DATA ANUS, SAWN, SWAN, AUNT, TUNA, WANT, YUAN, NAVY

31388 DATA ANY, YAWN, PROA, SOAP, ATOP, ROAR, SOAR, SORA
 31389 DATA ROTA, TARO, OSSA, OAST, STOA, AUTO, AVOW, PUPA
 31390 DATA PARR, RASP, SPAR, PART, RAFT, TARP, TRAP, WARP
 31391 DATA WRAP, PRAY, PASS, FAST, SPAT, UPAS, SWAP, WASP
 31392 DATA SPAY, PATY, YAUP, STAR, TSAR, SURA, TART, WART
 31393 DATA ARTY, TRAY, VARY, AWRY, WARY, SASS, TASS, VAST
 31394 DATA SWAT, WAST, WALE, WEAL, AMEN, MANE, MEAN, NAME
 31395 DATA MARE, REAM, MESA, SAME, SEAM, MATE, MEAT, META
 31396 DATA TAME, TEAM, AEON, NAFE, NEAP, PANE, PEAN, EARN
 31397 DATA NEAR, SANE, ANTE, NEAT, NAVE, VANE, VENA, ANEW
 31398 DATA WEAN, YEAN, PARE, PEAR, RAPE, REAP, APSE
 31399 DATA PATE, PEAT, TAPE, PAVE, RARE, REAR, ARSE, RASE
 31400 DATA SEAR, SERA, RATE, TARE, TEAR, UREA, AVER, RAVE
 31401 DATA WARE, WEAR, AERY, EYRA, YARE, YEAR, EAST, SATE
 31402 DATA SEAT, SETA, SAVE, VASE, EASY, EYAS, TEAT, UVEA
 31403 DATA AVE, GAFF, RAFF, WAFF, FLAG, FANG, HALF
 31404 DATA FASH, HAFT, KAIF, ALIF, FAIL, FILA, FAIN, NAIF
 31405 DATA FAIR, FIAT, WAIF, FLAK, FALL, FLAM, FLAN, FOAL
 31406 DATA LOAF, FLAP, FLAT, FLAW, FLAY, FOAM, FARM, FAUN
 31407 DATA FAWN, FARO, FORA, SOFA, OFAY, FRAP, FART, FRAT
 31408 DATA FRAY, ZARF, FAST, TUFA, WAFT, TANG, AGOG
 31409 DATA HANG, GASH, SHAG, GHAT, GAIN, AGIO, GAIT, VAGI
 31410 DATA KAGO, KAGU, GAWK, GALL, GOAL, SLAG, OGAM, GAMP
 31411 DATA GRAM, GAUM, GAMY, AGON, PANG, GNAR, RANG, SANG
 31412 DATA GNAT, TANG, GUAN, VANG, GNAW, YANG, SAGO, GOAT
 31413 DATA YOGA, GASP, QUAG, GUAR, RUGA, GRAY, GAST
 31414 DATA STAG, SWAG, YUGA, HASH, HATH, HAIK, HAIL, HILA
 31415 DATA HAIR, LAKH, ANKH, HANK, KHAN, KAPH, HARK, HAWK
 31416 DATA HALL, HALM, HALO, HARL, LASH, HALT, LATH, HAUL
 31417 DATA HULA, HYLA, HARM, MASH, SHAM, MATH, WHAM, HANT
 31418 DATA OPAH, HOAR, HORA, OATH, WHOA, AHOY
 31419 DATA HARP, HASP, PATH, RASH, HART, RATH, SASH, HAST
 31420 DATA SHAW, WASH, ASHY, SHAY, THAT, THAW, WHAT, ILIA
 31421 DATA KAKI, KAIL, KAMI, AKIN, PIKA, RAKI, SAKI, KIVA
 31422 DATA MAIL, ANIL, LAIN, NAIL, CHAT, TACH, CHAW, LAIC
 31423 DATA MICA, PICA, CALK, LACK, PACK, CARK, RACK
 31424 DATA CASK, SACK, TACK, CALL, CALM, CLAN, COAL, COLA
 31425 DATA LOCA, CLAP, CARL, TALC, CAUL, CLAY, LACY, COMA
 31426 DATA CAMP, CRAM, MARC, SCAM, CYMA, CARN, NARC, SCAN
 31427 DATA CANT, CAPO, COAT, TACO, CARP, CRAP, FACT, SCAR
 31428 DATA CART, CRAW, CAST, SCAT, TACT, CAVY, DEAD
 31429 DATA DADO, DUAD, DYAD, DEAF, FADE, AGED, EGAD, HEAD
 31430 DATA AIDE, IDEA, DALE, DEAL, LADE, DAME, MADE, MEAD
 31431 DATA DEAN, DARE, DEAR, READ, DATE, DEVA, AWED, WADE
 31432 DATA DAFF, FADO, FARD, DAFT, GADI, GLAD, DAGO, GOAD
 31433 DATA DRAG, GRAD, DHAK, HAND, HARD, DASH, SHAD

```

31434 DATA KADI, DIAL, LAID, AMID, MAID, PAID, QAID, ARID
31435 DATA RAID, DAIS, SAID, ADIT, AVID, DIVA, WADI, DANK
31436 DATA DARK, LAND, LOAD, LARD, AULD, DUAL, LADY, DAMN
31437 DATA MAND, DAMP, DRAM, DUMA, MAUD, DARN, NARD, RAND
31438 DATA SAND, DAWN, WADE, SODA, TOAD, PARD, SARD, DART
31439 DATA DRAT, DURA, DRAW, WARD, DRAY, YARD, DAWT, DAVY
31440 DATA AGEE, ALEE, EASE, FAKE, FEAL, FLEA, LEAF, FAME
31441 DATA FANE, FARE, FEAR, SAFE, FATE, FEAT, GAGE, GALE
31442 DATA GAME, MAGE, GAPE, PEAG, AGER, GEAR, RAGE
31443 DATA SAGE, GATE, GAVE, VEGAN, WAGE, HAKE, HALE
31444 DATA HEAL, AHEM, HAME, HEAP, HARE, HEAR, HATE, HEAT
31445 DATA HAVE, YEAH, KALE, LAKE, LEAK, MAKE, PEAK, RAKE
31446 DATA SAKE, TAKE, TEAK, WAKE, WEAK, WEKA, LEAL, ALME
31447 DATA LAME, MALE, MEAL, ELAN, ALOE, OLEA, LEAP, PALE
31448 DATA PEAL, PLEA, PERL, LEAR, RALE, REAL, HALE, SEAL
31449 DATA LATE, TAEI, TALE, TEAL, TELA, LAVE, VALE, VEAL
31450 DATA ABBE, BABA, ALBA, PACA, CASA, ACTA, DATA, AREA
31451 DATA HAAF, ALFA, AFAR, GAGA, ALGA, GALA, AGMA, ANGA
31452 DATA AGAR, RAGA, SAGA, AMAH, HAAR, AYAH, ARIA, KAKA
31453 DATA KANA, ARAK, ALMA, LAMA, ALAN, ALAR
31454 DATA ALAS, LAVA, MAMA, MANA, MAYA, ANNA, ANSA, ANTA
31455 DATA PAPA, PARA, TAPA, AURA, VARA, AWAY, ABBE, BABE
31456 DATA BLAB, BARB, BABU, BABY, BACH, BACK, CRAB, SCAB
31457 DATA ABED, BADE, BEAD, BALD, BAND, BARD, BRAD, DRAB
31458 DATA DAUB, BAWD, BAKE, BEAK, ABLE, BALE, BLAE, BEAM
31459 DATA BEMA, BANE, BEAN, BARE, BEAR, BRAE, BASE, ABET
31460 DATA BATE, BEAT, BETA, BEAU, ABYE, BARF, GAMB, BANG
31461 DATA BRAG, GARB, GRAB, GABY, BLAH, BASH, BATH, BAIL
31462 DATA IAMB, BAIN, BANI, ABRI, BIAS, BAIT, BALK, BANK
31463 DATA BARK, BASK, BALL, BALM, LAMB, SLAB, BLAT, BAWL
31464 DATA ABLY, AMBO, BARM, BARN, BRAN, BOAR, BORA, BOAT
31465 DATA BRAT, BRAW, BRAY, BASS, BAST, STAB, SWAB, ABUT
31466 DATA TUBA, ABUT, COCA, ACED, CADE, DACE, ACID, CADI
31467 DATA CAID, CLAD, CODA, CARD, SCAD, CAFE, FACE, CAGE
31468 DATA ACHE, EACH, CAME, ALEC, LACE, ACME, CAME, MACE
31469 DATA ACNE, CANE, CAPE, FACE, ACRE, CARE, RACE, CASE
31470 DATA CATE, TACE, CAVE, CALF, FACT, CLAG, CRAG, SCAG, ZZ

```

Listing 12-2. Word Square program.

```

1 GOSUB 9000:REM WORD SQUARE BY GEORGE SCHWENK (C) 1984
  TAB BOOKS INC.
2 IFLD=0THENLD=1:PRINT"LOADING VOCABULARY FILE":
  LOAD"VOCABL.DAT",8,1
4 POKE 52,64:POKE 56,64:CLR:GOTO100

```



```

10 REM COMPUTERS GET  A WORD
15 GOSUB 800: MV=0: MW$=""
20 SYS49184: REM GOTO GETWORD
30 REM GET WORD FROM RAM
40 WD$="": WL=PEEK(49153): FOR I=49154 TO 49153+WL:
WD$=WD$+CHR$(PEEK(I)): NEXT
50 IF WD$="ZZ" THEN RETURN
60 GOSUB 500: GOSUB 90: GOTO 20
90 PRINT "{HOME}{C/DN}{C/RT}{C/RT}";: PRINT MID$(TI$,3,2)":
"RIGHT$(TI$,2): RETURN
100 DIM LV(26), PT(98)
110 RESTORE
120 GOSUB 20000: REM INITIALIZATION
130 GOSUB 19000: REM GET GAME OPTIONS
140 GOSUB 3000: REM PRINT LETTER VALUES
190 GOSUB 22000: REM CLEAR ARRAYS
200 REM MAIN GAME LOOP
210 FOR NT=1 TO TT
220 FOR I=1 TO 5: BX$(I)="      ": NEXT I
230 GOSUB 12000: REM DISPLAY
240 GOSUB 4000: REM GENERATE RANDOM LETTERS
250 REM SET TIMER
260 TI$="000000"
270 GOSUB 1000: REM GET COMPUTER WORD
275 GET A$: IF ASC(A$+CHR$(0))=95 THEN 290
280 GOSUB 90: IF VAL(TI$)<DU*100 THEN 275
290 FQ=34334: GOSUB 41000: GOSUB 4600
300 REM MAIN PLAYER LOOP
310 FOR NP=1 TO TP
320 GOSUB 2000: REM GET PLAYER NP'S WORD
330 TS(NP)=SR*NW
340 NEXT NP
350 FOR I=1 TO 5: BX$(I)=CW$(I): NEXT I
360 GOSUB 11000: GOSUB 4700: REM DIPLAY COM WORD
365 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}"TAB(9)"{RVON}THE BEST THE WIZ CAN DO{RVDF}"
370 FOR I=0 TO 5
380 BS(I)=TS(I): CS(I)=BS(I)+CS(I)
390 NEXT I
400 GOSUB 10500: GOSUB 10600: GOSUB 10700: REM PRINT SCORE
440 GOSUB 2900: PRINT TAB(8)"HIT ANY KEY TO CONTINUE"
450 GETIO$: IF IO$="" THEN 450
460 NEXT NT
470 GOSUB 15000: REM FINAL SUMMARY
480 GETIO$: IF IO$="" THEN 480
490 IF IO$="Y" THEN 110

```

```

495 SYS 64738
499 END
500 REM EVALUATE WORD
510 V=0:FOR I=1 TO WL
520 V=V+LV(ASC(MID$(WD$,I,1))-64)
530 NEXT I
540 IF V>MV THEN MV=V:MW$=WD$
550 RETURN
600 REM GET INPUT WORD FROM PLAYER
610 PRINT "{RVON}"PY$(NP){RVOF}";
620 WD$="":INPUT WD$
630 RETURN
700 REM DELAY
710 FORD=1TO750:NEXTD
720 RETURN
800 POKE 251,0:POKE 252,64
810 RETURN
1000 REM COMPUTER DECISION
1010 FORI=1 TO5:CW$(I)="      ":NEXT I
1014 REM PASS CHRACTERS FOR MACHINE CODE
1015 FOR I=1 TO 5:POKE 49159+I,ASC(MID$(DI$,I,1)):NEXT I
1016 FOR I=1 TO 7:POKE 49167+I,ASC(MID$(LT$,I,1)):NEXT I

1020 SR=0:NW=0:FOR BW=1 TO 5
1030 POKE 49152,BW
1070 GOSUB 10:REM GOTO MACHINE CODE FOR WORD GET
1080 IF MV=0 THEN 1105
1090 NW=NW+1:CW$(BW)=MW$
1100 SR=SR+MV
1105 REM
1110 NEXT BW
1120 TS(0)=SR*NW
1130 RETURN
2000 REM ACCEPT INPUT WORD FROM COMPUTER
2010 REM CHECK IT AND ASK FOR VERIFICATION
2020 SR=0:NW=0
2030 FOR I=1 TO 5:PW$(I)="      ":NEXT I
2040 FOR I=1 TO 5:BX$(I)="      ":NEXT I
2050 FOR BW=1 TO 5
2060 GOSUB2900:REM CLEAR WINDOW & POSITON CURSOR
2070 PRINT " PLEASE ENTER YOUR WORD":PRINT "      ";
2080 FOR I=1 TO 5
2090 IF BW=I THEN PRINTMID$(DI$,BW,1);:GOTO2110
2100 PRINT "-";
2110 REM
2120 NEXT I:PRINT

```

```

2130 GOSUB 600:REM GET INPUT FROM PLAYER
2140 WL=LEN(WD$):REM CHECK WORD LENGTH
2150 IF WL=0 THEN 2295
2160 IF WL<BW THEN GOSUB2900:PRINT "WORD IS TOO SHORT":
GOSUB700:GOTO 2060
2170 IO$=MID$(DI$,BW,1)
2180 IF IO$<>MID$(WD$,BW,1) THEN GOSUB2900:PRINT "MUST USE
(RVON)"IO$(RVDF) AS #"BW:GOSUB700:
2190 GOSUB 2500:REM CHECK IF WORD IS POSSIBLE
2200 IF E=1 THEN GOSUB2900:PRINT "INVALID WORD":
GOSUB 700:GOTO2060
2210 GOSUB 2900:PRINT"(RVON)WORD AS ENTERED(RVDF) ";WD$
2220 PRINT " (RVON)V(RVDF)ALID WORD (RVON)T
(RVDF)RY AGAIN"
2230 PRINT " (RVON)I(RVDF)NVALID WORD SCORE=0
GET NEXT WORD"
2240 GETIO$:IF IO$="" THEN2240
2250 T=ASC(IO$)
2260 IF T=86 THEN 2300
2270 IF T=84 THEN 2060
2280 IF T=73 THEN 2295
2290 GOTO 2240
2295 REM
2296 NEXT BW:RETURN
2300 REM A GOOD WORD EVALUATE IT
2310 GOSUB500:NW=NW+1
2320 SR=SR+V
2330 PW$(BW)=WD$
2340 BX$(BW)=WD$
2350 GOSUB 11000:GOSUB10700
2360 GOTO 2295
2500 REM CHECK WORD VALIDITY
2510 PI$=LT$
2520 L=LEN(WD$)
2530 E=0
2540 FOR I=1 TO L:IF I=BW THEN 2600
2550 K=1
2560 FOR J=1 TO 7
2570 IF MID$(WD$,I,1)=MID$(PI$,J,1) THEN GOSUB 2700
2580 NEXT J
2590 IF K=1 THEN E=1
2600 NEXT I
2610 RETURN
2700 PI$=LEFT$(PI$,J-1)+" "+RIGHT$(PI$,7-J):K=0:J=7
2710 RETURN
2900 REM CLEAR WINDOW AND POSITION CURSOR

```

```

2910 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN} "
2920 FOR I=1 TO 3:PRINT "
NEXT I
2925 FOR I=1 TO 20:NEXT I
2930 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN} "
2940 RETURN
3000 REM PRITN LETTER VALUES
3010 PRINT "{CLR}"
3020 PRINT TAB(9)"{RVON}POINT VALUES OF LETTERS{RVOF}"
3030 FOR I=10 TO 1 STEP -1
3040 IF I=9 OR I=7 OR I=6 THEN 3105
3050 PRINT:PRINT:PRINT I;
3060 IF I<>10 THEN PRINT " ";
3070 PRINT " POINT LETTERS ";
3080 FOR J=1 TO 26
3090 IF LV(J)=I THEN PRINT CHR$(J+64);" ";
3100 NEXT J
3105 REM
3115 NEXT I
3120 PRINT:PRINT:PRINT:PRINTTAB(7)"{RVON}HIT ANY KEY TO
BEGIN GAME {RVOF}"
3130 GET IO$:IF IO$="" THEN 3130
3140 RETURN
4000 REM GENERATE DI$ & LT$
4005 LT$="":DI$="":PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}
"TAB(8);
4010 FOR I=1 TO 7
4020 GOSUB 4500
4030 PRINT IO$;"{C/RT}{C/RT}{C/RT}";
4040 FO=SC(I):GOSUB 41000
4050 LT$=LT$+IO$
4060 NEXT I
4100 FOR I=1 TO 5
4110 GOSUB4500
4120 DI$=DI$+IO$
4130 NEXT I
4200 REM PRINT DIAGONAL
4210 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{YEL0}"
4230 FOR I=1 TO 5
4240 PRINT TAB(12)MID$(DI$,I,1)"{C/DN}{C/DN}{C/RT}{C/RT}
{C/RT}";

```

```

4245 FQ=S2(I):GOSUB 41000
4250 NEXT I:PRINT"{WHT}";
4500 T=PT(INT(RND(O)*98))+64:IO#=CHR$(T)
4510 RETURN
4600 REM CLEAR LETTERS
4610 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}"TAB(8);
4620 FOR I=1 TO 7
4630 PRINT " {C/RT}{C/RT}{C/RT}";
4640 NEXT I
4650 RETURN
4700 REM PRINT LETTERS
4710 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{YELO}"TAB(8);
4720 FOR I=1 TO 7
4730 PRINT MID$(LT$,I,1){C/RT}{C/RT}{C/RT}";
4740 NEXT I:PRINT"{WHT}";
4750 RETURN
9000 REM SETCOLOR AND PRINT TITLE
9010 POKE 53281,0:PRINT "{LRED}":POKE53280,2
9020 PRINT "{RVON}{CLR}{C/DN}{C/DN}{RVON}"TAB(15)"WORD
SQUARE{RVDF}"
9030 PRINT:PRINT TAB(12)"BY GEORGE SCHWENK"
9040 PRINT:PRINT:PRINTTAB(8)"(C) 1984 TAB BOOKS INC."
9050 PRINT TAB(10)"ALL RIGHTS RESERVED"
9060 PRINT "{WHT}":RETURN
10500 REM PRINT INITIALS
10510 PRINT"{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}":PRINT"{LRED}"TAB(3)"WIZ"TAB(34)"WIZ":PRINT"{CYAN}"
10515 FOR I=1 TO TP
10520 PRINT TAB(3)PY$(I)TAB(34)PY$(I):PRINT
10530 NEXT I:PRINT"{WHT}"
10599 RETURN
10600 REM PRINT SCORES
10610 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}":PRINT"{LRED}";
10620 FOR I=0 TO TP:IF I=1THEN PRINT"{CYAN}";
10630 PRINT TAB(2)BS(I)TAB(33)CS(I):PRINT
10640 NEXT I:PRINT "{WHT}";
10650 RETURN
10700 REM PRINT DIAGONAL
10710 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{YELO}"
10720 FOR I=1 TO 5
10730 PRINT TAB(12)MID$(DI$,I,1){C/DN}{C/DN}{C/RT}{C/RT}
{C/RT}";
10740 NEXT I:PRINT"{WHT}";
10750 RETURN

```

```

10800 REM PRINT BOX
10810 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{YEL0}"
10820 FOR I=1 TO 5:PRINT TAB(12):FOR J=1 TO 5
10830 PRINT MID$(BX$(I),J,1){C/RT}{C/RT}{C/RT}";
10840 NEXT J:PRINT:PRINT:NEXT I:PRINT "{WHT}";
10850 RETURN
11000 REM PRINT BOARD
11010 PRINT"{CLR}";:PRINT "    {RVON}TIME{RVDF}
{RVON}LIMIT{RVDF}"TAB(30){RVON}TURN{RVDF}"
11020 PRINT "    0:00    ";STR$(DU);":00"TAB(28)NT"OF"TT
11030 PRINT TAB(17){RVON}LETTERS{RVDF}";"{LRED}"
11040 PRINT TAB(6) " |-----| "
11060 PRINT TAB(6) " | | | | | | | | " "
11080 PRINT TAB(6) " |-----| "
:PRINT:PRINT"{LBLU}"
11090 PRINT TAB(10) " |-----| "
11100 PRINT TAB(10) " | | | | | | | "
11110 PRINT TAB(10) " |-----+-----+-----+-----+-----| "
11120 PRINT TAB(10) " | | | | | | | "
11130 PRINT TAB(10) " |-----+-----+-----+-----+-----| "
11140 PRINT TAB(10) " | | | | | | | "
11150 PRINT TAB(10) " |-----+-----+-----+-----+-----| "
11160 PRINT TAB(10) " | | | | | | | "
11170 PRINT TAB(10) " |-----+-----+-----+-----+-----| "
11180 PRINT TAB(10) " | | | | | | | "
11190 PRINT TAB(10) " |-----+-----+-----+-----+-----| ":PRINT"
{WHT}"
11200 PRINT"{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
TAB(3){RVON}BOX{RVDF}"TAB(33){RVON}TOTAL{RVDF}"
11210 PRINT TAB(2){RVON}SCORE"TAB(33)"SCORE{RVDF}"
11220 GOSUB 10800:REM PRINT CURRENT BOX
11999 RETURN
12000 REM DO ENTIRE DISPLAY
12100 PRINT "{CLR}":GOSUB11000:GOSUB 10500:GOSUB 10600:
RETURN
15000 REM END OF GAME SUMMARY
15010 PRINT "{CLR}{C/DN}{C/DN}"TAB(15){RVON}FINAL SUMMARY"
15020 PRINT:PRINTTAB(7){RVON}PLAYER","SCORE","RATING{RVDF}"
15030 PRINT TAB(9)"WIZ",INT(CS(0)):PRINT
15040 FOR I=1 TO TP
15050 R=CS(I)/CS(0):R=R*R*R
15060 RA=INT(R*6-DU*2/10)
15070 IF RA>5 THEN RA=5
15080 IF RA<0 THEN RA=0
15090 RA=6-RA

```

```

15100 PRINT TAB(9)PY$(I),INT(CS(I)),RT$(RA)
15110 NEXT I
15120 PRINT "{C/DN}{C/DN}{C/DN}{C/DN}"TAB(10)"ANOTHER GAME?
(Y/N)"
15130 RETURN
19000 REM OPTION PHASE
19010 PRINT "{CLR}"
19020 PRINT "{RVON}OPTION PHASE{RVOF}"
19030 PRINT:PRINT "ENTER # OF PLAYERS (1-5)";
19040 INPUT TP
19050 IF TP<1 OR TP>5 THEN 19030
19060 FOR I=1 TO TP
19070 PRINT:PRINT "ENTER 3 INITIALS FOR PLAYER #"I;
19080 INPUT IO$:IF LEN(IO$)<>3 THEN 19070
19090 PY$(I)=IO$
19100 NEXT I
19110 PRINT:PRINT"ENTER # OF TURNS TO PLAY";
19120 INPUT TT
19130 IF TT<1 THEN 19110
19140 PRINT:PRINT "ENTER # OF MINUTES PER TURN (1-10)";
19150 INPUT DU
19160 IF DU<1 OR DU>10 THEN 19140
19170 PRINT:PRINT "{RVON}SELECTIONS{RVOF}":PRINT:PRINT
19180 PRINT "PLAYERS"
19190 FOR I=1 TO TP
19200 PRINT I;PY$(I)
19210 NEXT I
19220 PRINT:PRINT "NUMBER OF TURNS ";TT
19230 PRINT:PRINT "MINUTES PER TURN";DU
19240 PRINT:PRINT "ARE THESE SELECTIONS OK (Y/N)";
19250 GET IO$:IF IO$="" THEN 19250
19260 IF IO$<>"Y" THEN 19000
19270 REM CLEAR SCORES
19280 FOR I=0 TO TP
19290 CS(I)=0
19300 BS(I)=0
19310 RETURN
20000 REM PROGRAM INITIALZATION
20010 PRINT "{CLR}"
20020 PRINT "PLEASE WAIT INITIALIZING PROGRAM"
20030 FOR I=1 TO 26
20040 READ LV(I)
20050 NEXT I
20060 FOR I=0 TO 97
20070 READ PT(I)
20080 NEXT I

```

```

20100 REM READ IN RATINGS
20110 FOR I=1 TO 6
20120 READ RT$(I)
20130 NEXT I
20140 REM READ IN SCALE VALUES
20150 FOR I=1 TO 7
20160 READ SC(I)
20170 NEXT I
20180 REM SCALE 2
20190 FOR I=1 TO 5
20200 READ S2(I)
20210 NEXT I
21000 GOSUB 49110:REM READ IN GETWORD SUBROUTINE
21999 RETURN
22000 REM CLEAR ARRAYS
22010 FOR I=0 TO 5
22020 BS(I)=0:CS(I)=0
22030 NEXT I
23000 REM INIT
23010 GOSUB 40000
29999 RETURN
30000 REM LETTER VALUES
30010 DATA 1,3,3,2,1,4,2,4,1,8,5,1,3,1,1,3,10,1,1,1,1,4,4,
8,4,10
30020 REM PROBABILITY TABLE
30030 DATA 1,1,1,1,1,1,1,1,2,2,3,3,4,4,4,4,5,5,5,5,5,5,5,
5,5,5,6,6
30040 DATA 7,7,7,8,8,9,9,9,9,9,9,9,9,10,10,11,12,12,12,12,
13,13
30050 DATA 14,14,14,14,14,14,15,15,15,15,15,15,15,15,16,16,
17,17,18,18,18
30060 DATA 18,18,18,19,19,19,19,20,20,20,20,20,20,20,21,21,
21,21
30070 DATA 22,22,23,23,24,24,25,25,26,26
30100 REM RATINGS
30110 DATA WIZARD,GENIUS,MASTER,EXPERT,STUDENT,NOVICE
30200 REM SCALE VALUES
30205 DATA 2145,2408,2703,2864,3215,4050,4291
30210 REM SCALE 2
30220 DATA 4291,3215,2864,2408,2145
40000 REM SOUND INIT
40010 S=54272:FOR I=STOS+24:POKE I,0:NEXT
40020 DR=160
40030 POKE S+5,9:POKE S+6,0
40040 POKE S+24,15
40050 RETURN

```



```

41000 REM PLAY FQ ON VOICE 1
41010 HF=INT(FQ/256):LF=FQ-HF*256
41020 POKE S+1,HF:POKE S,LF
41030 POKE S+4,33
41040 FORD=1TODR:NEXTD
41050 POKE S+4,32
41060 FORD=1T050:NEXT D
41070 RETURN
49110 I=49152
49120 READ A:IF A=256 THEN RETURN
49130 POKE I,A:I=I+1:GOTO 49120
49152 DATA 0,0,0,0,0,0,0,0
49160 DATA 0,0,0,0,0,0,0,0
49168 DATA 0,0,0,0,0,0,0,0
49176 DATA 0,0,0,0,0,0,0,0
49184 DATA 160,0,177,251,48,6,153,2
49192 DATA 192,200,16,246,41,127,153,2
49200 DATA 192,200,152,141,1,192,24,101
49208 DATA 251,133,251,165,252,105,0,133
49216 DATA 252,173,2,192,201,90,208,8
49224 DATA 173,3,192,201,90,208,1,96
49232 DATA 173,1,192,205,0,192,48,200
49240 DATA 172,0,192,136,185,2,192,217
49248 DATA 8,192,208,188,56,144,185,160
49256 DATA 0,185,16,192,153,24,192,200
49264 DATA 192,7,208,245,160,0,200,204
49272 DATA 0,192,208,6,204,1,192,208
49280 DATA 2,96,136,185,2,192,162,0
49288 DATA 221,24,192,240,8,232,224,7
49296 DATA 208,246,24,144,208,169,0,157
49304 DATA 24,192,200,204,1,192,208,214
49312 DATA 96,256

```

use to form words.

K—Error flag. If K=1, then an invalid word has been detected.

J—Loop counter.

DI\$—Holds the five diagonal letters contained in the word square.

R—Ratio of players' final scores to that of the Wiz.

RA—Calculated rating number that is used to index RT\$.

S—= 54272, which is the first address in the C-64 sound (SID) chip.

DR—Duration loop limit used in the sound-playing routine.

HF—High byte of frequency of sound to be played by the sound subroutine at 41000.

LF—Low byte of frequency of sound to be played by the sound subroutine at 41000.

WORD SQUARE ARRAY VARIABLES

The following array variables are used in Word Square. The numbers in parentheses are the array dimensions of the variables.

LV (26)—Contains the letter value of each letter. It is indexed by the ASCII value of a letter minus 64.

PT (98)—Probability table used to choose the random letters. It contains the ASCII value of a letter minus 64. It is indexed by a random number from 1 to 98.

BX\$ (5)—Contains words in the square. It is indexed by the variable BW.

CW\$ (5)—Contains the computer's words for the square. It is indexed by the variable BW.

TS (5)—Contains the total score for a square and equals $NW * SR$. It is indexed by NP, where $NP = 0$ corresponds to the Wiz.

BS (5)—Contains the total score for the current turn for each player. It is indexed by NP.

CS (5)—Contains the cumulative score of all turns up to and including the present turn for each player. It is indexed by NP.

PY\$ (5)—Contains three initials for each player. It is indexed by NP.

PW\$ (5)—Contains the word input by the player to be placed in the square. It is indexed by BW.

RT\$ (6)—Contains the ratings designations: Wizard, novice, etc. It is indexed by RA.

SC (7)—Contains the frequencies of the notes that are played when random letters are displayed. It is indexed by a loop counter that runs from 1 to 7.

S2 (5)—Contains the frequencies of the notes that are played when random letters in the diagonal of the word square are displayed. It is indexed by a loop counter that runs from 1 to 5.

WORD SQUARE SUBROUTINES

The following subroutines are used in Word Square:

10—Contains the routine used by the computer to determine the highest-scoring word for a given position in the square. On entry, BW equals the position in the square. On exit, MW\$ contains the best word and MV contains its value. If $MV = 0$, then the computer could not find a valid word.

500—Calculates the sum of letter values for the word contained in WD\$ and stores the sum in

variable V. If V is greater than MV, then MW\$ is set equal to WD\$.

600—Accepts input of a word from a player.

700—Time delay used for message displays.

800—Resets the vocabulary pointer to the beginning of the word list.

1000—Called to determine the computer's words for this turn. At the start, it sets CW\$ to all blanks. On exit, TS(0) equals the Wiz's score for this square, and CW\$ contains the chosen words.

2000—Gets input words from players and checks for validity. At the start, PW\$ and BX\$ are filled with blanks. On exit, PW\$ and BX\$ contain the players' words, NW contains the number of words, and SR contains the sum of the letter values of these words.

2300—Calls word-evaluation routine, sets PW\$ and BX\$ equal to WD\$, and adds the word's letter sum to the sum of all the words' letter values ($SR = SR + V$).

2900—Clears input window and positions the cursor.

3000—Displays all 26 alphabet letters and their individual scoring values.

4000—Generates and displays the random letter sequence contained in PI\$ and LT\$.

4200—Displays DI\$ in the diagonal.

4600—Clears display of LT\$ letters.

4700—Prints LT\$ letters on the display screen without sound and delay.

9000—Sets colors for display and prints the title page.

10500—Prints players' initials on the screen display.

10600—Prints box scores and total scores on the screen display.

10700—Prints the diagonal on the screen display.

10800—Prints all the letters in the square that are currently contained in BX\$.

11000—Prints the display board on the screen.

12000—Calls the various subroutines necessary to print an entire display.

15000—Calculates player ratings and displays the end-of-game summary.

19000—Gets and sets game options.

20000—Reads program data and initializes program variables and arrays.

40000—Initializes the sound chip for note playing.

41000—Plays voice 1 with the frequency contained in FQ.

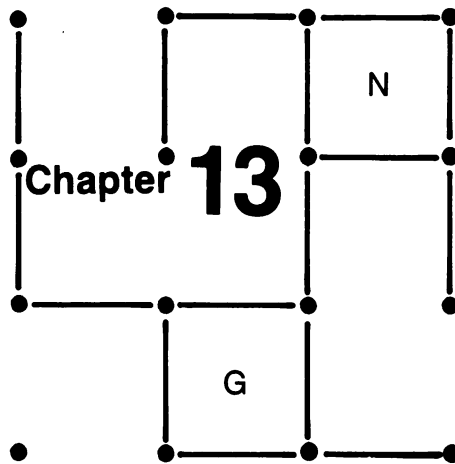
WORD SQUARE PROGRAM DESCRIPTION

The Word Square program begins with a call to the subroutine at line 9000 to set colors and to print the title page. This is followed by the loading of the vocabulary file, which is protected by the POKes in line 4. This part of the program is executed only when the program is first run. The main flow of the program begins in line 100 with the dimensioning of the program arrays. The rest of the program initialization phase is contained in lines 120-190. Subroutine calls are made to line 20000 to read and initialize program variables, to line 19000 to get game options, to line 3000 to print the list of letters and their scoring values, and finally to line 22000 to clear arrays.

The main turn, or control, loop begins at line 200 with a FOR-NEXT loop that sets the number of turns (NT) to run from 1 to the total number of turns (TT) that were selected in the game option subroutine. At the beginning of each turn, the

display is redrawn (GOSUB 12000), and a new set of letters is determined and displayed (GOSUB 4000). After all the letters are displayed, the system timer is reset, and the subroutine at line 1000 is called to determine all five of the computer's (Wiz's) words. This routine takes about 3-5 seconds, after which the program enters a loop until the time limit chosen in the option phase is reached. Then another FOR-NEXT player loop is set to run from 1 to the total number of players (TP) chosen in the option phase. Next, five words from each player are accepted (GOSUB 2000), and their scoring value is determined (SR*NW).

After the last player's words are entered, the Wiz's words and all the players' scores are displayed (GOSUB 11000, GOSUB 4700, GOSUB 10500, GOSUB 10600, GOSUB 10700). This ends the turn, and the same procedure is repeated for each turn until NT is greater than TT. At the end of the game, the rating for each player is determined and displayed by the subroutine at line 15000. Finally, the players are given a chance to replay the game. If a replay is desired, control returns to line 110, where the DATA read pointer is restored. The title page, vocabulary-file loading, and array dimensioning, which are done before line 110, are thus not repeated.



Takeover Program Documentation

This chapter gives a detailed description of the program Takeover, which is presented in Listing 13-1. First, we define the use of all variables, arrays, and subroutines used in the program. We follow this with a very simple flow-chart for the entire program (Fig. 13-1) and then go through the main program line by line, highlighting the important subroutines and their functions.

TAKEOVER VARIABLES

The following variables are used in Takeover:

NT—Current turn number.
 FT—First turn number with which to begin this game.
 TT—Total turns in the game.
 NP—Index number of current active player.
 TP—Total number of players in the game.
 SF—A selection flag. If it equals 1, then the last corporate balance sheet requested is displayed.
 SK—Equals a corporate number (1-20). It is used to index the various arrays that contain corporate data.

LC—Index number of the last corporation for which a check sheet command was made.

IR—Current interest rate percentage.

I—Loop counter.

T—Temporary storage variable.

FG—Flag used for printing a balance sheet display. If FG=0, the sheet is printed on the left side of the screen. If FG=1, the sheet is printed on the right side of the screen.

E—Error flag. If E=1, then an error has been detected.

I\$—Used to accept input from keyboard.

IO\$—Used to hold a numeric string for formatted output.

L—Length of IO\$, also used as a loop counter.

AJ—Adjustment factor used to modify stock prices for block transactions with the market.

AU—Adjusted price of a stock.

TS—Total shares outstanding for a corporation.

SM—Sum variable used together with SI to total the value of investment securities.

J—Loop counter.

SI—Sum variable used together with SM to total the value of investment securities.

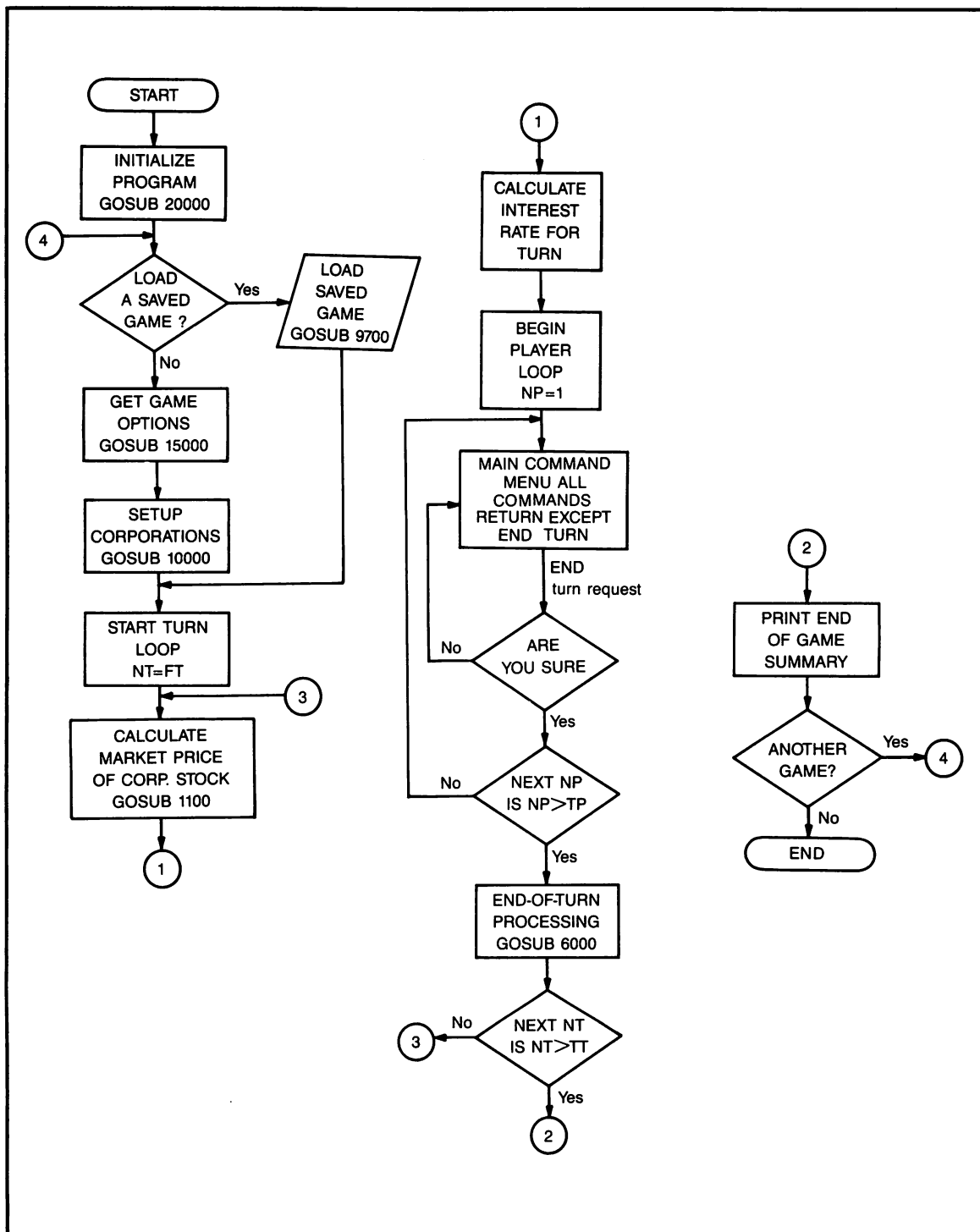


Fig. 13-1. The flowchart for the Takeover program.

Listing 13-1. Takeover program.

```

1 GOTO 100
9 GOTO 520
100 GOSUB 20000:REM INITIALIZE PROGRAM
101 PRINT "{C/DN}{C/DN}{C/DN}DO YOU WISH TO LOAD A SAVED
GAME(Y/N)":GOSUB700
103 IF T=89 THENE=0:GOSUB9700:GOTO190
110 GOSUB 15000:REM GET GAME OPTIONS
120 GOSUB 10000:REM INITIALIZE CORPORATIONS
190 IF E=1 THEN E=0:GOTO101
200 FOR NT=FT TO TT
250 GOSUB 1100:REM CALCULATE PRICES
260 IR=IR+INT(RND(O)*4-RND(O)*4):REM CALCULATE INTEREST
RATE
262 IF IR<5 THEN IR=5
264 IF IR>20 THEN IR=20
270 PRINT "{CLR}":GOSUB 2000
300 FOR NP=1 TO TP
310 REM CLEAR TRANSACTION FLAG
320 FOR I=1 TO20:TF(I)=0:NEXT I
450 SF=0
470 GOSUB 3100:SK=NP:FG=0:GOSUB 2500
475 IF SF=1 THEN SK=LC:FG=1:GOSUB 2500
500 REM MAIN MENU
510 GOSUB 900:REM PRINT MENU
520 GOSUB 700:T=T-64
530 IF T<1 OR T>20 THEN 520
540 ONTGO9,4000,4200,1500,550,9,9,9,4300,9,9,9,9,9,9,
8500,4400,4700,7000,5000
550 REM END TURN
560 GOSUB 3000:REM CLEAR WINDOW
570 PRINT "DO YOU REALLY WISH TO END TURN (Y/N)"
580 GOSUB 700:IF T<>89 THEN 510
585 REM YES END TURN
590 NEXT NP:GOSUB 6000:REM END OF TURN PROCESSING
595 IF NT=TT THEN 645
600 PRINT "{CLR}"
610 PRINT "{RVOF}SUMMARY END OF TURN #{RVOF} "NT" OF "TT
620 PRINT:PRINT:GOSUB 11000
630 PRINT:PRINT:PRINT "DO WISH TO SAVE GAME (Y/N)"
635 GOSUB 700:IF T<>89 THEN 640
636 GOSUB 9600:PRINT:PRINT "DO YOU WISH TO END GAME (Y/N)"
:GOSUB 700
638 IF T=89 THEN END
640 PRINT:PRINT "CALCULATING NEW MARKET PRICES"

```

```

645 NEXT NT
650 PRINT "{CLR}"
660 PRINT "{RVON}END OF GAME SUMMARY{RVOF}"
670 PRINT:PRINT:GOSUB 11000
680 PRINT:PRINT:PRINT"DO YOU WISH TO PLAY AGAIN (Y/N)
690 GOSUB 700
695 IF T=89 THEN 101
699 END
700 REM GET A KEY
710 GET I$:IF I$="" THEN 710
720 T=ASC(I$)
730 RETURN
750 REM GET CORP LETTER
760 PRINT "PRESS KEY TO SELECT CORPORATION/STOCK"
770 GOSUB700
775 IF T=92THEN RETURN
778 T=T-64
780 IF T<1 OR T>20 THEN 770
790 RETURN
800 REM PRINT IO$ RIGHT JUSTIFIED IF FIELD OF 4
810 L=LEN(IO$):IF L<5 THEN IO$=" "+IO$:GOTO810
820 PRINT MID$(IO$,2,4);
830 RETURN
850 REM DELAY ROUTINE
860 TI$="000000"
870 IF TI<300 THEN 870
880 RETURN
900 GOSUB 3000:REM PRINT MENU
910 PRINT "{RVON}COMMAND MENU PRESS KEY TO SELECT{RVOF}";
:PRINT
920 PRINT "{RVON}P{RVOF}URCHASE {RVON}S{RVOF}ELL {RVON}Q
{RVOF}UOTE {RVON}D{RVOF}EBT PAYOFF"
930 PRINT "{RVON}C{RVOF}HECK SHEET {RVON}I{RVOF}NVEST
{RVON}B{RVOF}ORROW      IR=";
935 IO$=STR$(IR):GOSUB 800:PRINT "%"
940 PRINT "{RVON}R{RVOF}OI {RVON}T{RVOF}AKEOVER {RVON}
E{RVOF}ND TURN      TURN=";NT;
950 RETURN
1000 REM CALCULATE PRICE QUOTE ADD ADJ FACTOR
1010 REM FG=0 FOR BUY =1 FOR SELL
1020 AJ=1+BK/MS(SK)
1030 IF FG=0 THEN QU=PR(SK)*AJ:GOTO1050
1040 QU=PR(SK)/AJ
1050 RETURN
1100 REM EVALUATE STOCK PRICES
1105 GOSUB 1200:REM CALCULATE SI

```

```

1110 FOR I=1 TO 20
1115 TS=OS(I)+MS(I)+PS(I)
1117 IF TS=0 THEN PR(I)=0:GOTO 1130
1120 PR(I)=INT((CH(I)+IV(I)+SI(I)-DB(I))*100/TS)
1130 NEXT I:RETURN
1200 REM CLACULATE SI
1210 FOR I=1 TO 20
1220 SM=0
1230 FOR J=1 TO 5
1240 IF IS(J,I)=0 THEN 1270
1260 SM=SM+IS(J,I)*PR(SO(I,J))/100
1270 NEXT J:SI=SM
1280 NEXT I
1290 RETURN
1500 GOSUB 3000:PRINT "{RVON}PAYING OFF DEBT{RVOF}"
1510 PRINT "ENTER AMOUNT":INPUT P:IF P>CH(NP) THEN PRINT "NOT
ENOUGH CASH":GOSUB 850:GOTO 1500
1520 DB(NP)=DB(NP)-P:CH(NP)=CH(NP)-P:GOTO 470
2000 REM OUTPUT MARKET BLOCK
2010 PRINT "{HOME}{RVON}      PRICE      PRICE      PRICE
PRICE"
2020 FOR I=0 TO 4:FOR J=1 TO 4
2030 K=4*I+J
2050 Z=INT((J-1)*10)
2060 PRINT TAB(Z){RVON}"CP$(K)"{RVOF}  "INT(PR(K));
2070 NEXT J:PRINT:NEXT I
2090 RETURN
2500 REM OUTPUT CORPORATE BALANCE SHEET
2530 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}"TAB(FG*20);
2540 PRINT "      "CP$(SK)"      ROI="INT(RI(SK))"%"
2550 PRINT TAB(FG*20);
2560 IO$=STR$(CH(SK))
2570 PRINT "CASH ";:GOSUB 800
2575 PRINT " DEBT ";
2580 IO$=STR$(DB(SK))
2590 GOSUB 800:PRINT
2595 PRINT TAB(FG*20);
2600 IO$=STR$(IV(SK))
2610 PRINT "IVST ";:GOSUB 800
2615 PRINT TAB(FG*20+10)"STOCK SHS"
2620 PRINT TAB(FG*20+11)"OWN ";
2630 IO$=STR$(OS(SK)):GOSUB 800:PRINT
2640 PRINT TAB(FG*20+11)"PLY ";
2650 IO$=STR$(PS(SK)):GOSUB 800:PRINT
2660 PRINT TAB(FG*20+11)"MRK ";

```



```

2670 IO$=STR$(MS(SK)):GOSUB800:PRINT
2680 PRINT:PRINT TAB(FG*20+10)"$$$ @";PR(SK)
2700 IO$=STR$(PR(SK)*(OS(SK)+PS(SK)+MS(SK))/100)
2710 PRINT TAB(FG*20+11)" =";
2720 GOSUB 800
2730 IF SK<=TP THEN GOSUB 2800
2790 RETURN
2800 REM PRINT INVESTMENT SECURITIES OWNED
2805 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}";
2810 FOR II=1 TO 5
2820 Q=SO(II,SK):IF Q=0 THEN 2850
2830 PRINT TAB(FG*20)CP$(Q);" ";
2840 IO$=STR$(IS(II,SK)):GOSUB 800
2850 PRINT:NEXT II
2860 RETURN
3000 REM CLEAR INPUT WINDOW
3010 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}";
3011 PRINT "
3012 PRINT "
3013 PRINT "
3014 PRINT "
3020 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}";
3030 RETURN
3100 REM CLEAR MID WINDOW
3105 PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}"
3110 FOR II=1 TO 13
3115 PRINT "
3120 RETURN
4000 REM BORROW MONEY
4010 GOSUB 3000
4020 PRINT TAB(10){RVON}BORROW MONEY{RVOF}"
4030 PRINT "ENTER AMOUNT IN MILLIONS";
4040 INPUT B:IF B+DB(NP)>CH(NP)+IV(NP)-DB(NP) THEN 4100
4045 IF B=0 THEN GOTO500
4050 IF B<0 THEN 4000
4060 DB(NP)=DB(NP)+B
4070 CH(NP)=CH(NP)+B
4080 GOTO 470
4100 GOSUB 3000:PRINT "YOU ARE UNDER CAPITALIZED."
4110 PRINT "THE BANK DENIES YOUR APPPLICATION."
4140 GOSUB 850:GOTO 470

```

```

4200 REM CHECK BALANCE SHEET
4210 GOSUB3000:SF=1
4220 GOSUB 750:REM GET CORP KEY
4230 IF T=92 THEN 500
4260 LC=T:GOTO 470
4300 REM INVESTMENT
4310 GOSUB3000:PRINTTAB(10)"{RVON}INVESTING MONEY{RVOF}"
4330 PRINT "ENTER AMOUNT IN MILLIONS";
4340 INPUT M:IF M=0 THEN 500
4350 IF M>CH(NP) THEN 4390
4360 IV(NP)=IV(NP)+M
4370 CH(NP)=CH(NP)-M
4380 GOTO 470
4390 PRINT "YOU DON'T HAVE ENOUGH CASH.":GOSUB 850:
GOTO4300
4400 REM PRICE QUOTE
4410 GOSUB3000:GOSUB 750
4420 IFT=92 THEN 500
4425 SK=T
4426 GOSUB 3000:PRINT"HOW MANY BLOCKS";
4427 INPUT BK:IF BK=0 THEN 500
4430 GOSUB 3000:PRINT "HIT KEY":PRINT "{RVON}B{RVOF}UY
QUOTE {RVON}S{RVOF}ELL QUOTE"
4440 GOSUB700:IF T=92 THEN 500
4445 IF T=66 THENFG=0:GOSUB 1000:GOTO4600
4450 IF T=83 THEN FG=1:GOSUB1000:GOTO4500
4460 GOTO 4430
4500 PRINT"{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}"TAB(20)"{RVON}SELL QUOTE: ";
4510 PRINT CP$(SK); "{RVOF}"
4520 PRINT TAB(18); IO$=STR$(BK):GOSUB800
4530 PRINT " @ ";
4540 IO$=STR$(QU):GOSUB 800
4545 IO$=STR$(QU*BK/100):PRINT " =":GOSUB 800:PRINT" MIL";
4550 GOTO500
4600 PRINT"{HOME}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}{C/DN}"TAB(20)"{RVON}BUY QUOTE: ";CP$(SK); "
{RVOF}"
4610 GOTO 4520
4700 GOSUB 3100:PRINT "{HOME}{C/DN}{C/DN}{C/DN}{C/DN}
{C/DN}{C/DN}":PRINT "{RVON}%RETURN ON INVESTMENT FOR
EACH CORP"
4710 C=0:FOR I=1 TO5:FOR J=0 TO 3
4720 C=C+1

```

```

4730 PRINT TAB(J*10)CP$(C);" ";
4740 REM
4750 IO$=STR$(RI(C)):GOSUB800
4760 NEXTJ:PRINT:NEXT I
4770 GOTO 500
5000 REM TAKEOVER ATTEMPT
5010 GOSUB 3000:PRINT "{RVON}ATTEMPTING TAKEOVER{RVOF}"
5020 GOSUB 750
5030 IF T=92 THEN 500
5040 IFT<=TPTHENPRINT"YOU CAN NOT TAKEOVER A MAJOR CORP":
GOSUB850: GOTO5000
5050 REM CHECK IF YOU ARE MAJORITY STOCKHOLDER
5060 SK=T:GOSUB9000
5070 PE=IS(NS,NP)/(MS(SK)+PS(SK))
5080 IF PE>.5 THEN 5115
5090 PRINT "YOU TAKEOVER BID FAILS"
5100 GOSUB850:GOTO500
5115 GOSUB3000:PRINT "OK - PROCESSING TRANSACTION"
5120 REM CALCULATE TOTAL VALUE OF MINORITY INTEREST
5130 TM=CH(SK)+IV(SK)-DB(SK)
5140 REM CALCULATE NORMALIZED ROI
5145 IT=IV(NP)+IV(SK)
5150 RI(NP)=IV(NP)*RI(NP)/IT+IV(SK)*RI(SK)/IT
5160 REM COMBINE COPRS
5170 CH(NP)=CH(NP)+CH(SK)
5180 IV(NP)=IT
5190 DB(NP)=DB(NP)+DB(SK)
5200 REM CALCULATE TOTAL VALUE OF NEW CORP
5210 GOSUB1200:VC=CH(NP)+IV(NP)-DB(NP)+TM*PE
5220 REM CALCULATE NEW NUMBER OF SHARES TO ISSUE
5230 OT=OS(NP)+MS(NP)+PS(NP):R=TM/VC
5240 SN=R*OT/(1.-R)
5250 REM CALCULATE ADDITIONAL SHARES OWNERS
5260 REM OF LIQUIDATED CORP WILL RECEIVE
5270 FS=SN/(PS(SK)+MS(SK))
5280 REM DISTRIBUTE NEW SHARES
5410 FOR L= 1 TO TP
5420 IF L=NP THEN5480
5430 TEMP=NP:NP=L:GOSUB9000
5440 IF NS=0 THEN 5470
5450 BK=IS(NS,L)*FS:CO=TEMP:IS(NS,L)=0:SO(NS,L)=0
5460 GOSUB 8000:REM ADD BK OF CO TO NP
5465 PS(TEMP)=PS(TEMP)+BK
5470 NP=TEMP
5480 NEXT L
5490 REM NOW DO MARKET DISTRIBUTION

```

```

5500 MS(NP)=MS(NP)+MS(SK)*FS
5510 REM CLEAR STOCK
5520 CH(SK)=0:DB(SK)=0:IV(SK)=0
5530 MS(SK)=0:RI(SK)=0:OS(SK)=0:PS(SK)=0
5540 GOSUB 9000:IS(NS,NP)=0:SD(NS,NP)=0
5550 GOSUB 1100:REM NEW STOCK PRICES
5560 GOSUB1100:PRINT"{CLR}":GOSUB2000:GOTO450
6000 REM END OF TURN PROCESSING
6002 PRINT "{CLR}{RVON}END OF TURN PROCESSING{RVOF}"
6005 GOSUB 1200:REM CALCULATE SI
6010 FOR I=1 TO 20
6020 REM CAL ROI AND ADD TO CASH
6030 CH(I)=CH(I)+RI(I)*IV(I)/100
6040 REM SERVICE DEBT
6050 DC=IR*DB(I)/100
6060 IF DC<CH(I) THEN 6100
6070 DF=DC-CH(I)
6080 IV(I)=IV(I)-DF
6090 CH(I)=CH(I)+DF
6100 CH(I)=CH(I)-DC
6110 IF I<=TP THEN 6190
6120 REM MINOR CORP INVESTS ALL CASH
6130 NI=CH(I)
6140 CH(I)=CH(I)-NI
6150 IV(I)=IV(I)+NI
6190 NEXT I
6200 RETURN
7000 REM SELLING STOCK
7010 GOSUB3000:PRINT"{RVON}SELLING STOCK{RVOF}"
7020 GOSUB 750
7030 IF T=92 THEN 500
7035 IF TF(T)=1 THEN 9500
7040 SK=T:GOSUB 9000
7050 IF NS<>0 THEN7600
7060 IF SK<>NPTHENPRINT"YOU DON'T OWN STOCK IN ";CP$(SK):
GOSUB850:GOTO7000
7070 GOSUB3000:PRINT"SELL HOW MANY BLOCKS OF ";CP$(SK);
7080 INPUT BK
7090 IF BK>OS(NP)THEN GOSUB3000:PRINT"YOU DON'T OWN THAT
MUCH":GOSUB850:GOTO7000
7100 GOSUB3000:PRINT"DO YOU WISH TO SELL TO"
7110 PRINT "{RVON}M{RVOF}ARKET {RVON}A{RVOF}NOTHER PLAYER"
7120 GOSUB 700
7130 IF T=92 THEN 500
7135 IF T=77 THEN 7400
7140 IF T<>65 THEN 7100

```

```

7150 GOSUB 3000
7160 GOSUB 750
7170 IF T=92 THEN 500
7175 IFT >TF OR T=NP THEN 7150
7180 BY=T:GOSUB3000:PRINTCP$(BY);" GIVE PRICE PER BLK";
7190 INPUT PR:TL=PR*BK/100
7200 GOSUB3000
7210 PRINTCP$(NP);" SELLS "BK"BLKS OF "CP$(SK)" TO ";
7300 PRINT CP$(BY);:PRINT " @":IO$=STR$(PR):GOSUB800:
PRINT" TOTAL COST=";
7310 IO$=STR$(TL):GOSUB800:PRINT" MIL."
7320 PRINT "VERIFY (Y/N)";:GOSUB 700
7330 IF T=92 THEN 500
7340 IF T=89 THEN 7360
7350 GOSUB3000:PRINT "REJECTED":GOSUB850:GOTO7000
7360 IFCH(BY)<TLTHENPRINTCP$(BY);" DOESN'T HAVE ENOUGH
CASH":GOSUB 850:GOTO7000
7365 TEMP=NP:NP=BY:CO=SK:GOSUB8000:NP=TEMP:IF E=1 THEN 470
7370 CH(NP)=CH(NP)+TL
7375 PS(NP)=PS(NP)+BK
7380 CH(BY)=CH(BY)-TL
7390 OS(NP)=OS(NP)-BK
7395 TF(SK)=1:GOTO470
7400 REM SELL STOCK IN OWN CORPORATION TO MARKET
7410 FG=1:GOSUB1000:TL=QU*BK/100
7420 GOSUB3000
7430 PRINT"SELL "BK" BLK(S) OF "CP$(SK);" TO MARKET @";
7440 IO$=STR$(QU)
7450 GOSUB800:PRINT
7460 PRINT"VERIFY (Y/N)"
7470 GOSUB700:IF T=92 THEN500
7475 IF T=89 THEN 7490
7480 GOSUB 3000:PRINT "REJECTED"
7490 REM PROCESSING TRANSACTION
7500 OS(NP)=OS(NP)-BK
7510 CH(NP)=CH(NP)+TL
7520 MS(SK)=MS(SK)+BK
7530 TF(SK)=1:GOTO470
7600 REM SELL INVESTMENT SECURITES
7610 GOSUB3000:PRINT"SELL HOW MANY BLK(S) OF "CP$(SK);
7620 INPUT BK:IF BK=0 THEN 500
7625 IF BK>IS(NS,NP)THENGOSUB3000:PRINT"YOU DON'T HAVE
THAT MUCH": GOSUB850:GOTO7000
7630 GOSUB3000:PRINT"DO YOU WISH TO SELL TO"
7640 PRINT "(RVON)M(RVOF)ARKET (RVON)A(RVOF)NOTHER PLAYER"
7650 GOSUB 700

```

```

7660 IF T=92 THEN 500
7670 IF T=77 THEN 7800
7680 IF T<>65 THEN 7600
7690 GOSUB 3000
7700 GOSUB 750
7710 IF T=92 THEN 500
7715 IFT>TF OR T=NP THEN 7690
7720 BY=T:GOSUB3000:PRINTCP$(BY);" GIVE PRICE PER BLK"
7730 INPUT PR:TL=PR*BK/100
7735 GOSUB3000
7740 PRINTCP$(NP);" SELLING "BK" BLOCKS OF "CP$(SK)" TO "
7750 PRINT CP$(BY);" @";:IO$=STR$(PR):GOSUB800:PRINT"
TOTAL COST=" ;
7760 IO$=STR$(TL):GOSUB800:PRINT" MIL."
7770 PRINT "VERIFY (Y/N)";:GOSUB 700
7772 IF T=92 THEN 500
7775 IF T=89 THEN 7780
7778 GOSUB3000:PRINT "REJECTED":GOSUB850:GOTO7000
7780 IFCH(BY)<TLTHENPRINTCP$(BY);" DOESN'T HAVE ENOUGH
CASH":GOSUB 850:GOTO7000
7781 IF SK=BY THEN 7791
7782 TEMP=NP:NP=BY:CO=SK:GOSUB8000:NP=TEMP:IF E=1 THEN 470
7784 CH(NP)=CH(NP)+TL
7786 CH(BY)=CH(BY)-TL
7788 IS(NS,NP)=IS(NS,NP)-BK:IF IS(NS,NP)=0 THEN SO(NS,NP)=0
7790 TF(SK)=1:GOTO470
7791 CH(NP)=CH(NP)+TL
7792 OS(BY)=OS(BY)+BK
7793 CH(BY)=CH(BY)-TL
7795 IS(NS,NP)=IS(NS,NP)-BK:IF IS(NS,NP)=0 THEN SO(NS,NP)=0
7796 TF(SK)=1:GOTO470
7800 REM SELL —
ESTMENT STOCK TO MARKETLIST 7175
7810 FG=1:GOSUB1000:TL=QU*BK/100
7820 GOSUB3000
7830 PRINT "SELL "BK" BLK(S) OF "CP$(SK)" TO MARKET":
PRINT "@";
7840 IO$=STR$(QU)
7850 GOSUB800:PRINT
7860 PRINT "VERIFY (Y/N)";
7870 GOSUB 700:IFT=92 THEN 500
7875 IFT=89 THEN 7890
7880 GOSUB3000:PRINT"REJECTED":GOSUB850:GOTO7000
7886 CH(BY)=CH(BY)-TL
7890 REM PROCESSING TRANSACTION
7900 IS(NS,NP)=IS(NS,NP)-BK:IF IS(NS,NP)=0 THEN SO(NS,NP)=0

```

```

7910 CH(NP)=CH(NP)+TL
7920 MS(SK)=MS(SK)+BK
7930 PS(SK)=PS(SK)-BK
7940 TF(SK)=1:GOTO470
8000 REM ADD BK TO CO OF NP
8010 REM
8020 E=1
8030 FOR II=1 TO 5
8040 IF SO(II,NP)=CO THEN IS(II,NP)=IS(II,NP)+BK:E=0:II=5
8050 NEXT II
8060 IF E=0 THEN RETURN
8070 FOR II=1 TO 5
8080 IF SO(II,NP)=0 THEN SO(II,NP)=CO:IS(II,NP)=BK:E=0:II=5
8090 NEXT II
8100 IF E=1 THEN GOSUB 3000:PRINT"YOU MAY ONLY OWN 5 MINOR
CORPS":GOSUB850
8110 RETURN
8300 REM PURCHASING STOCK
8510 GOSUB 3000:PRINT " {RVON}PURCHASING STOCK{RVOF}"
8520 GOSUB750
8525 IF T=92 THEN 500
8530 SK=T:CO=SK
8540 IF TF(CO)=1 THEN 9500
8550 GOSUB 3000
8560 PRINT "FROM WHOM DO YOU WISH TO PURCHASE"
8570 PRINT " {RVON}M{RVOF}ARKET {RVON}A{RVOF}NOTHER
PLAYER"
8580 GOSUB700
8585 IF T=92 THEN 500
8590 IF T=77 THEN 8850
8600 IF T<>65 THEN 8550
8610 GOSUB 3000:PRINT " {RVON}PURCHASING STOCK FROM ANOTHER
PLAYER{RVOF}"
8620 GOSUB 750
8630 IF T=92 THEN 500
8634 IF T>TP THEN 8610
8640 IFT=NPTHEN GOSUB3000:PRINT"YOU CAN'T BUY FROM
YOURSELF":GOSUB850:GOTO8610
8645 IFT=STKTHEN GOSUB3000:PRINT"MAJOR CORP STOCK MUST BE
OFFERED":GOSUB850:GOTO8500
8650 SL=T:SL$=CP$(SL)
8652 TEMP=NP:NP=SL
8655 GOSUB9000:NP=TEMP:IFNS=0THEN GOSUB3000:PRINTSL$;"DOES
NOT OWN STOCK":GOSUB850:GOTO8610
8660 GOSUB 3000:PRINT " {RVON}PURCHASING STOCK{RVOF}"
8670 PRINT "IN "CP$(SK)" FROM "SL$

```

```

8680 PRINT "ENTER BLKS TO BUY,PRICE";
8690 INPUT BK,PR
8700 IF BK=0 THEN 500
8710 IF BK>IS(NS,SL) THEN GOSUB 3000:PRINT SL$" DOES NOT HAVE
      THAT MUCH":GOSUB 850:GOTO 8610
8720 IF BK*PR/100>CH(NP) THEN GOSUB 3000:PRINT "NOT ENOUGH
      CASH":GOSUB 850:GOTO 8610
8730 GOSUB 3000:PRINT "{RVON}TRANSACTIONS AS ENTERED{RVOF}"
8740 PRINT "BUY "BK" BLK(S) OF "CP$(SK);" FOR";
8750 IQ$=STR$(PR):GOSUB 800:PRINT
8760 PRINT SL$" IS THIS ACCEPTABLE (Y/N)"
8770 GOSUB 700:GOSUB 3000
8780 IF T=89 THEN PRINT "ACCEPTED":GOSUB 850:GOTO 8790
8785 PRINT "REJECTED":GOSUB 850:GOTO 8500
8790 REM PROCESSING TRANSACTION
8792 IF SK=NP THEN 8832
8795 GOSUB 8000
8800 CH(NP)=CH(NP)-BK*PR/100
8810 CH(SL)=CH(SL)+BK*PR/100
8820 IS(NS,SL)=IS(NS,SL)-BK
8825 IF IS(NS,SL)=0 THEN SQ(NS,SL)=0
8830 TF(SK)=1:GOTO 470
8832 OS(NP)=OS(NP)+BK
8834 CH(NP)=CH(NP)-BK*PR/100
8835 CH(SL)=CH(SL)+BK*PR/100
8836 IS(NS,SL)=IS(NS,SL)-BK
8837 IF IS(NS,SL)=0 THEN SQ(NS,SL)=0
8838 PS(NP)=PS(NP)-BK
8840 TF(SK)=1:GOTO 470
8850 GOSUB 3000:PRINT "{RVON}PURCHASING FROM THE MARKET
      {RVOF}"
8900 PRINT "ENTER BLK(S) OF "CP$(SK)" TO PURCHASE";
8910 INPUT BK
8915 IF BK=0 THEN 500
8920 FG=0:GOSUB 1000
8930 TC=QU*BK/100:IF TC>CH(NP) THEN GOSUB 3000:PRINT "YOU
      CAN'T AFFORD":GOSUB 850:GOTO 8850
8935 GOSUB 3000:PRINT "OK - PROCESSING TRANSACTION"
8938 IF SK=NP THEN 8975
8940 GOSUB 8000
8950 CH(NP)=CH(NP)-TC
8960 MS(SK)=MS(SK)-BK
8965 PS(SK)=PS(SK)+BK
8970 TF(SK)=1:GOTO 470
8975 OS(SK)=OS(SK)+BK
8980 MS(SK)=MS(SK)-BK

```



```

8985 CH(NP)=CH(NP)-TC
8990 TF(SK)=1:GOTO470
9000 REM SET NS= TO INDEX 0 IF NOT FOUND
9010 NS=0
9020 FORI=1 TO 5
9030 IF SK=SO(I,NP) THEN NS=I
9040 NEXT I:RETURN
9500 GOSUB 3000:PRINT "ONLY 1 TRANSACTION PER CORP PER
TURN"
9510 GOSUB 850:GOTO470
9600 REM SAVE GAME
9605 PRINT:PRINT "{RVON}SAVING GAME{RVOF}"
9610 PRINT:PRINT "ENTER FILE NAME":INPUT F$
9620 OPEN 5,8,5,"O:"+F$+"",S,W"
9630 IF ST<>0 THEN 9900
9640 FOR I=1 TO 20
9650 PRINT#5,CH(I)
9651 PRINT#5,IV(I)
9652 PRINT#5,DB(I)
9653 PRINT#5,PS(I)
9654 PRINT#5,MS(I)
9655 PRINT#5,OS(I)
9656 PRINT#5,RI(I)
9660 FOR J=1 TO 5
9670 PRINT#5,IS(J,I)
9671 PRINT#5,SO(J,I)
9680 NEXT J:NEXT I
9681 PRINT#5,NT+1
9682 PRINT#5,TT
9683 PRINT#5,TP
9685 IF ST<>0 THEN 9900
9690 CLOSE 5:RETURN
9700 REM RETRIEVE GAME
9705 PRINT:PRINT "{RVON}RETRIEVING GAME{RVOF}"
9710 PRINT:PRINT "ENTER FILE NAME":INPUT F$
9720 OPEN 5,8,5,"O:"+F$+"",S,R"
9730 IF ST<>0 THEN 9900
9740 FOR I=1 TO 20
9750 INPUT#5,CH(I)
9751 INPUT#5,IV(I)
9752 INPUT#5,DB(I)
9753 INPUT#5,PS(I)
9754 INPUT#5,MS(I)
9755 INPUT#5,OS(I)
9756 INPUT#5,RI(I)
9760 FOR J=1 TO 5

```

```

9770 INPUT#5,IS(J,I)
9771 INPUT#5,SO(J,I)
9780 NEXT J:NEXT I
9781 INPUT#5,FT
9782 INPUT#5,TT
9783 INPUT#5,TP
9785 IF ST<>0 AND ST<>64 THEN 9900
9790 CLOSE 5:RETURN
9900 PRINT:PRINT "I/O TERMINATED BY ERROR #"ST:CLOSE5
9910 GOSUB850:E=1
9920 RETURN
10000 REM INITIALIZE CORP VALUES
10010 REM CH,IV,OS,MS,PS,RI,SO
10012 PRINT:PRINTTAB(10)"<<< PLEASE WAIT >>>"
10015 PRINT:PRINTTAB(8)"{RVON}SETTING UP CORPORATIONS
{RVDF}"
10020 FOR I=1 TO TP
10030 CH(I)=10*RND(0)+20
10040 IV(I)=20*RND(0)+40
10050 DB(I)=CH(I)+IV(I)-50
10060 RI(I)=10.0
10070 FOR J=1 TO 5
10080 IS(J,I)=0:SO(J,I)=0
10090 NEXT J
10100 OS(I)=60:PS(I)=0:MS(I)=40
10110 NEXT I
10120 FOR I=TP+1 TO 20
10130 CH(I)=1+10*RND(0)
10140 IV(I)=1+10*RND(0)
10150 DB(I)=(CH(I)+IV(I))*(RND(0)/2)
10160 RI(I)=10+40*RND(0)
10170 SI(I)=0
10200 OS(I)=0:PS(I)=0:MS(I)=50
10205 FOR J=1 TO 5:IS(J,I)=0:SO(J,I)=0:NEXT J
10210 NEXT I
10220 GOSUB 1100:REM INITIAL EVALUATION OF STOCK PRICES
10230 RETURN
11000 REM PRINT VALUE SUMMARY
11010 PRINT "{RVON}CORP","VALUE","OWNER WORTH{RVDF}"
11020 FOR I=1 TO TP
11030 V=SI(I)+CH(I)+IV(I)-DB(I)
11040 W=OS(I)/(OS(I)+PS(I)+MS(I)):W=W*V
11050 IO$=STR$(V)
11060 PRINT CP$(I),:GOSUB 800:PRINT ,
11070 IO$=STR$(W):GOSUB 800:PRINT
11080 NEXT I:RETURN

```

```

15000 REM GAME OPTIONS
15010 PRINT "{CLR}"
15020 PRINT TAB(10) "{RVON}SELECT GAME OPTIONS{RVOF}"
15030 PRINT:PRINT:PRINT "ENTER # OF PLAYERS";
15040 INPUT I$:TP=VAL(I$):IF TP<1 OR TP>10 THEN 15030
15050 PRINT:PRINT:PRINT "ENTER # OF TURNS";
15060 INPUT I$:TT=VAL(I$):IF TT<1 THEN 15040
15070 PRINT:PRINT:PRINT "OPTIONS AS SELECTED"
15080 PRINT:PRINT "NUMBER OF PLAYERS=";TP
15090 PRINT:PRINT "NUMBER OF TURNS=";TT
15100 PRINT:PRINT "ARE THESE SELECTIONS OK (Y/N)":GOSUB 700
15110 IF T<>89 THEN 15010
15120 FT=1:RETURN
20000 DIM CH(20),IV(20),CP$(20),DB(20),PR(20),DS(20),
MS(20),SQ(5,20),TF(20)
20010 DIM PS(20),IS(5,20),RI(20),SI(20)
20020 PRINT "{CLR}{C/DN}{C/DN}{RVON}"TAB(16)"TAKEOVER{RVOF}"
20022 PRINT "{C/DN}{C/DN}"TAB(19)"BY":PRINT "{C/DN}"
TAB(13)"GEORGE SCHWENK"
20024 PRINT "{C/DN}{C/DN}"TAB(9)"(C) 1984 TAB BOOK CO."
20026 PRINTTAB(10)"ALL RIGHTS RESERVED"
20030 FOR I=1 TO 20
20040 READ CP$(I)
20050 NEXT I
20060 LC=20:IR=12
20070 BL$=""
20099 RETURN
30000 REM CORP NAMES
30010 DATA ATT,BIC,CBS,DEC,EPC,FAC,GAC,HBA,IBM,JJI,KCC,
LOC,MMM,NBC,DEC,PFR,QOF
30020 DATA RAP,SOC,TUS

```

P—Amount of debt in millions of dollars a player requests to pay off.

K—Temporary value variable.

Z—Temporary value variable.

Q—Temporary value variable.

II—Inner loop counter.

B—Amount of money in millions of dollars a player wants to borrow.

M—Amount of money in millions of dollars a player wants to invest.

C—Corporate index number used in same way as SK.

PE—The fraction of the total stock outstanding

in a corporation that a player owns at the time he or she attempts a takeover of this corporation.

TM—Total value of minority interest in a corporation that is being taken over.

IT—Investment total.

VC—Total value of a new corporation after a merger with a corporation that has been taken over.

OT—Total outstanding shares.

R—Ratio of the minority interest to the value of a new corporation ($R=TM/VC$).

FS—Fractional shares of a new corporation that owners of the taken-over corporation receive per share of stock they own in the old corporation.

SN—Temporary value variable used to calculate FS.

BK—Blocks of stock. Each block equals 10,000 shares.

TEMP—Temporary storage for corporate index.

CO—Corporate index used for adding BK investment shares.

DC—Cost of interest in millions of dollars on debt financing for this turn.

DF—Difference in millions of dollars between DC and cash on hand.

NI—New investment in millions of dollars.

BY—Corporate index of a buyer.

TL—Total value of a transaction.

SL—Corporate index of a seller.

TC—Total value of a transaction.

NS—Investment securities index.

F\$—File name used to save or load a game.

BL\$—A variable containing 39 spaces.

TAKEOVER ARRAY VARIABLES

The following array variables are used in Takeover. The numbers in parentheses are the array dimensions of the variables.

CH(20)—Cash on hand in millions of dollars. It is indexed by corporate number.

IV(20)—Investment in millions of dollars. It is indexed by corporate number.

CP\$(200)—Three-letter symbol for each corporation. It is indexed by corporate number.

DB(20)—Current debt in millions of dollars. It is indexed by corporate number.

PR(20)—Market price in dollars of a single share of stock. It is indexed by corporate number.

OS(20)—Number of blocks (10,000 shares) held by principal owner of corporation. It is indexed by corporate number.

MS(20)—Number of blocks of stock outstanding on the open market. It is indexed by corporate number.

PS(20)—Number of blocks of stock held by other players in the game. It is indexed by corporate number.

SO(5,20)—Corporate index of stock owned for investment. The first index runs from 1 to 5, allow-

ing five investment securities to be owned. The second index is the corporate number of the owning corporation.

IS(5,20)—Used together with the SO array; it has the same indexes. It contains the number of blocks of investment stock held.

TF(20)—Transaction flag. If TF = 1, then a transaction has been made this turn. It is indexed by corporate number.

SI(20)—Total value in millions of dollars of investment securities. It is indexed by corporate number.

RI(20)—Return on investment of millions of dollars.

TAKEOVER SUBROUTINES

Note that some of the following line sequences are not called by a GOSUB-RETURN; instead, control is passed to the line sequences with a GOTO. Another GOTO returns control from these line sequences to the main menu. However, in all other ways these line sequences function as a subroutine.

700—Gets an input key from the keyboard and stores its ASCII value in I\$.

750—Gets an input key from the keyboard and converts it to its corporate index (1-20). Invalid key inputs are rejected.

800—Prints the contents of IO\$ right-justified in a field of four at the current cursor position.

850—Delay routine used for message display.

900—Prints out the command menu.

1000—Calculates the price-adjustment factor and price-per-share quotes for market transactions.

1100—Calculates market prices for all of the corporations.

1200—Calculates the total value of investment securities (SI) for each of the corporations.

1500—Handles the payoff debt command.

2000—Prints out the market prices of all the corporations.

2500—Prints out the corporate balance sheet for corporation number SK. If FG = 1, the sheet is displayed on the right side of the screen. If FG = 0, the sheet is displayed on the left side of the screen.

2600—Prints out the investment securities owned by corporation number SK. If FG = 1, they are displayed on the right side of the screen. If FG = 0, they are displayed on the left side of the screen.

3000—Clears the input window at the bottom of the screen and positions the cursor at the upper left hand corner of the window.

3100—Clears the balance sheet window.

4000—Handles the borrow money command.

4300—Handles the investment command.

4400—Handles the market quote command.

4700—Prints the display of the return on investment for each corporation.

5000—Processes the takeover command and subsequent liquidation of the corporation that has been taken over.

7000—Processes the selling of stock to both the market and other players.

8000—Adds BK blocks of corporation SK to the investment securities owned by player NP.

8500—Processes the buying of stock from both the market and other players.

9000—Sets NS to the first index of SO(5,20) when SO = SK.

9500—Displays the “only 1 transaction per turn” error message.

9600—Saves the game.

9700—Retrieves a saved game.

9900—Acts as an error handler to save and retrieve game routines.

10000—Initializes the corporate balance sheet values for CH, DB, etc.

15000—Accepts game options from players and sets appropriate variables accordingly.

20000—Dimensions arrays and prints the title screen.

TAKEOVER PROGRAM DESCRIPTION

The TAKEOVER program begins with a subroutine call to line 20000, where the program is initialized. Next the user is given the opportunity to load a saved game. If this option is chosen, the corporation data arrays are set to the values they had when the loaded game was saved, and the game

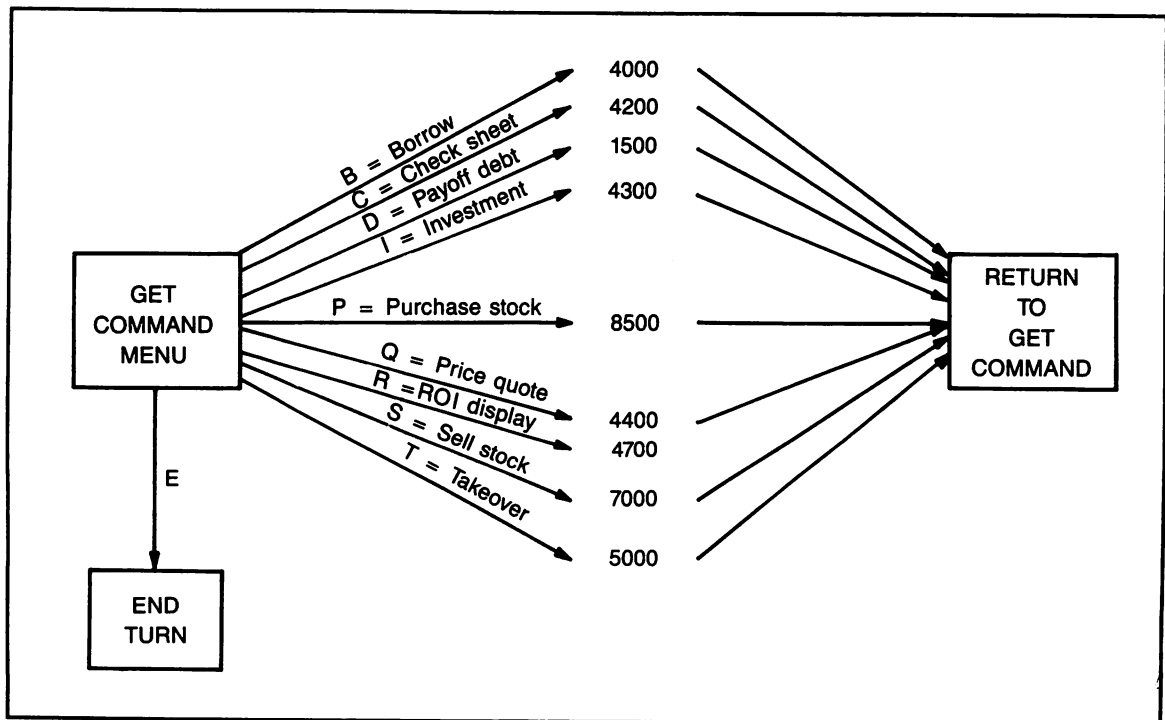


Fig. 13-2. The Takeover command menu.

option (GOSUB 15000) and corporation initialization routines are skipped. If the load-saved game option is not chosen, then a GOSUB 15000 asks the players for their desired game options. Then the corporate data arrays are randomly set with a GOSUB 10000.

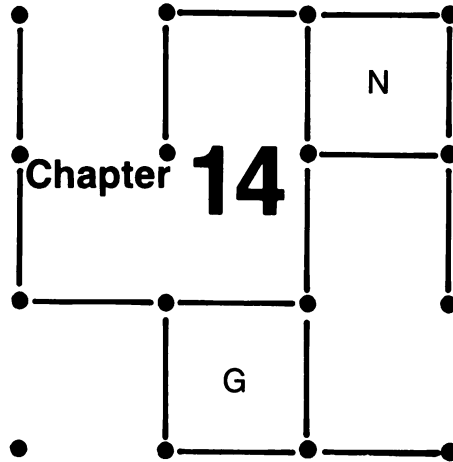
The main program control loop begins at line 200 with a FOR-NEXT loop that counts the number of turns (NT), which is set to run from FT to TT. FT is the first turn number, which equals 1 for new games or equals the value loaded from a saved game file. The first operation in the main loop is the setting of market prices for the turn by a GOSUB 1100. This is the only time in the turn loop that this subroutine is called unless a successful takeover is made, in which case the 1100 subroutine is called by the 5000 subroutine. After the market prices are set, the interest rate in effect for this turn is calculated. A subloop on player index NP is now begun. NP runs from 1 to the total number of players (TP), which was chosen in the option phase. At the start of each player's turn, the TF array is cleared. The TF array is set equal to 1 in the program whenever a transaction is performed with the stock of corporations. This flag is used to prevent the player from making more than one transaction

per turn per corporation.

After updating the display screen with subroutine calls to 3100 and 2500, the heart of the Takeover program, the main menu, is reached. All commands are executed by the ON GOTO statement here, and all the command routines, with the exceptions of the end-turn command, return to the main menu. Figure 13-2 shows this command structure.

When all players end their turns, the end-of-turn processing calculations are done by a GOSUB 6000, and then the end-of-turn summary is printed by a GOSUB 11000. The players are given the opportunity to save the game at this point. If this option is chosen, the necessary data arrays are written to disk by the 9600 subroutine. The main turn loop is now repeated until the total number of turns for the game is reached.

After exiting the main turn loop, a final turn summary is printed, and the players are given the opportunity to play another game. If this option is chosen, a GOTO line 101 is executed, and the players are given the choice of playing a saved game or starting a new one. By using a GOTO instead of a RUN statement, we save the time consumed by the program initialization routine at line 20000.



Making Money: Advice to Profiteers

If you want to put the knowledge gained from this book to profitable use, this chapter is dedicated to you. We give some additional advice on the game-design process from a professional point of view. We describe the steps you should go through to submit your game design to a publisher, and we describe the typical publishing contract. We describe what you should expect from such a contract, some typical pitfalls, and some attractive agreements in these contracts. In Appendix A, we list some game software publishers who purchase game designs from independent designers.

ADVICE TO THE NOVICE

While BASIC is adequate for most designs for your own use, commercial products require that you learn machine language. There is tremendous competition in the game-design field, and the performance advantage machine language possesses over BASIC is enough to make a product stand out among other games. Unfortunately, many publishers do not give a game a serious trial if it

is coded in BASIC, which they consider amateur. If you plan to create a professional game-design project, we reiterate our recommendation that you learn FORTH. FORTH is ideal for game design because it gives a good trade-off between programming effort and performance. Machine language routines are easily integrated into a FORTH program, and the source code is more easily transported to another microcomputer.

DO NOT quit your job to become 'a game designer! We are sure you have read many stories about the vast riches made by various superkids. However, these riches were made by very few designers and were made at a time when microcomputers were just coming on the market and little software was available. The sad fact is that many independent game designers are working at a minimum hourly wage, which they endure because they enjoy their work so much. So, do not embark on a professional game-design project simply to get rich; you might be severely disappointed with the results! Choose a project you will enjoy in an area about which you are knowledgeable. For example,

if you are a military officer, you can use your knowledge of military history and tactics to design a good war game. And if your game is accepted for publication, the money you receive will be “icing on the cake” of enjoyment you receive by seeing your own design come alive on the computer.

WRITING DOCUMENTATION

After you finish your game-design project, there are a few things you must do before you submit it to a publisher. First, prepare a clear and concise manual that describes how to load, start, and play the game. Give examples of what the player will see on the screen and fully describe the rules of play. Be sure to remember that many game players are only using their computers as an appliance and may not know anything about programming. On the other hand, do not write a novel. Game players are anxious to play the game and do not want to wade through a voluminous treatise.

After you complete the manual, distribute copies of the game and the manual to friends and relatives who have not seen your game and have no prior knowledge of how to play the game. This process is known as *blind testing*. The blind test should turn up bugs you may have missed in your program and omissions or ambiguities in your documentation. Consider all the comments your play testers make. Do not be offended by their criticisms or feel that all their recommendations must be heeded. Usually you will find some suggestions helpful, and you will make changes to the game prior to submitting it to a publisher.

SUBMITTING YOUR GAME TO A PUBLISHER

When your game program and documentation are complete, it is time to find a publisher. You might try some of those listed in Appendix A. Many publishers have independent-author kits that contain forms that should accompany your game, so the first step is to write to request such a kit if they have one available.

An important part of your game submission is the transmittal letter. The purpose of the letter is to get your program a serious review by the

publisher. Remember that the publisher is not primarily interested in the game for the game's sake, but is interested in making money. Therefore, your letter should stress the marketing aspects of your game. Explain why your game is unique and why it should do well in the marketplace. Identify competing games and describe why yours is superior and why it should be the preferred choice of consumers.

After you submit your game to a publisher, you can expect a response in 30 to 90 days. Don't hesitate to contact the publisher if you aren't contacted after 90 days. The response will come in one of three forms. One is an outright rejection. Another is a rejection with some suggested improvements and a request that you resubmit your game for reevaluation. Still another is an acceptance letter that is usually accompanied by a draft copy of a publishing contract.

If your game is rejected, do not be discouraged. If the rejection letter lists a contact for further information, call the publisher and try to determine the reasons your game was rejected. If no contact is listed and the letter doesn't give an explanation other than “your program does not meet out marketing objectives at this time,” it probably means that your game was given a cursory evaluation, and you should be leary of submitting material to this publisher in the future. If your game is rejected, but the publisher requests that you resubmit the game with changes, it means that the publisher has taken the time to evaluate your game and is genuinely interested in publishing it. If you can make the requested changes without completely rewriting the program, then it is well worth your effort to make the changes quickly and resubmit your game to this publisher.

If your game is accepted for publication, it is time to celebrate! But if you want to gain the maximum financial reward from your endeavor, you must be very careful about the contractual arrangements you make with the publisher. Although we cannot give you legal advice about such contracts (consult your lawyer!), we can explain some of the key conditions in such contracts and, based on experience, give our opinions about what to expect.

CONTRACT TERMS

Contracts differ from company to company; don't assume they're all the same. Read the contract carefully and make notes.

Publishing Rights

The two types of publishing rights that are normally granted to the publisher are exclusive and nonexclusive rights. Generally, exclusive rights grant the publisher the sole right to publish your game for the period of the contract, and you may not offer the game to another publisher. Nonexclusive rights allow you to offer your game to other publishers, usually under certain price or other conditions. If your program is accepted by a major publishing company, you should not be apprehensive about granting them exclusive rights. Such rights are reasonable if you expect them to make a large financial commitment to publishing your program. If you are giving rights to a small or unestablished publishing house, however, you should strive to get them to agree to nonexclusive publishing rights. This allows you the protection of being able to seek other publishers should your initial publisher fail to promote and sell your game satisfactorily.

Conversion Rights

Most game publishers attempt to offer their game products on the four major microcomputer systems—Apple, ATARI, Commodore, and IBM. Most contracts contain provisions that grant the publisher the right to convert your game to other computer systems. It is greatly to your benefit to grant the publisher these rights, which will increase your financial reward. But do not give these rights away too cheaply. Conversion royalties are normally given as a percentage of the royalties for the original version. The percentage you should receive for conversions depends on two factors: the programming difficulties involved in converting your game to another system, and the value of your name as the author of a similar game product. If you are a new author, you cannot expect any value for name recognition, so the percentage will depend almost entirely on the difficulty involved in converting your game for use on another computer. For arcade-style

games that require considerable programming, a 20 percent share for you is a reasonable amount. If your game required considerable research and development, such as would be necessary for a detailed simulation, you should require at least a 50 percent share of the royalties.

Royalties

You are normally compensated by the publisher in the form of royalty payments based on the actual number of games sold. These payments usually range from 10 percent of the wholesale price to 20 percent of the retail price, which is quite a spread. You must be very careful negotiating in this area, particularly if you have offers from more than one publisher. It is often more profitable to accept a contract at 10 percent of the wholesale price from a major established publisher than to accept 20 percent of the retail price from a new and under-financed company.

Advances

Most publishers are reluctant to give you any advances against future royalties. An advance is one of the most important clauses you should strive to obtain in a publishing agreement. An advance payment is greatly to your benefit for the following reasons:

1. If the publisher makes a financial commitment to your game, it is a good indication that they plan to promote your game actively and not simply list it in their catalog.
2. The market conditions in the microcomputer world change rapidly. What is hot today may be cold tomorrow. If a publisher holds your program too long before publishing it, they may change the amount of support they give your game, or, worse yet, continue to hold your program until you ask to be let out of the contract. A healthy advance is the best protection you have against this situation. It is reasonable to ask for an advance of \$2,000 to \$3,000, even for your first game design.

We hope you find our advice helpful, especially if you are new to professional game design. Should you attempt such a project, we offer you the best of luck for an enjoyable and profitable endeavor.

Appendix A

Software Producers

The following is a list of software producers who accept games for publication from free-lance authors. For a more complete listing you can refer to *Microprogrammer's Market 1984*, by Marshall Hamilton, published by TAB Books Inc.

Adventure International
Subsidiary of Scott Adams, Inc.
P.O. Box 3435
Longwood, FL 32750

Avalon Hill Microcomputer Games
4517 Harford Rd.
Baltimore, MD 21214

Broderbund Software
1938 Fourth St.
San Rafael, CA 94901

Datasoft, Inc.
19519 Business Center Dr.
Northridge, CA 91324

IBM Software Publishing Division
P.O. Box 1328, Dept. 2C6,232-2
Boca Raton, FL 33432

Synapse Software
5221 Central Ave.
Richmond, CA 94804

Appendix B

Listing Conventions

Most Commodore 64 programs contain code symbols that perform functions such as color changes and cursor movements. Unfortunately these codes may mean something different to your printer, so a direct listing of the program could result in a garbled output. To prevent this problem, the listings in this book use special listing symbols to represent the control characters. If you are typing in the pro-

grams contained in this book directly from the listings, then you may either replace the symbols with their CHR\$ form given below, or you may enter the required characters directly from the keyboard. For example, when you see SC you can replace it with CHR\$(147), or you can type “[SHIFT][CLR/HOME]” at your keyboard.

CHARACTER STRING	LISTING SYMBOL	SYMBOL DEFINITION	CHARACTER STRING	LISTING SYMBOL	SYMBOL DEFINITION
CHR\$(17)	CD	Cursor Down	CHR\$(148)	IN	Insert
CHR\$(18)	RV	Reverse On	CHR\$(150)	LR	Light Red
CHR\$(19)	HM	Home Cursor	CHR\$(151)	G1	Gray 1
CHR\$(28)	RD	Red	CHR\$(152)	G2	Gray 2
CHR\$(29)	CR	Cursor Right	CHR\$(153)	LG	Light Green
CHR\$(30)	GN	Green	CHR\$(154)	LB	Light Blue
CHR\$(31)	BL	Blue	CHR\$(155)	G3	Gray 3
CHR\$(129)	OR	Orange	CHR\$(156)	PU	Purple
CHR\$(144)	BK	Black	CHR\$(157)	CL	Cursor Left
CHR\$(145)	CU	Cursor Up	CHR\$(158)	YL	Yellow
CHR\$(146)	RO	Reverse Off	CHR\$(159)	CY	Cyan
CHR\$(147)	SC	Screen Clear/Home Cursor	CHR\$(160)	SS	Shifted Space

Index

A

Abstract strategy games, 27
Advances, 130
Advantages of computer games, 1
Adventure games, 27, 41
Algorithms for computer players, 41
Arcade games, 26, 41
Array variables, Prime Time, 73
Array variables, Takeover, 125
Array variables, Word Square, 106
Arrays, 64
Artificial intelligence models, 29
Artificial intelligence, 41
Assembly language, 60

B

Backgammon, 54
Balancing the contest, 23
Baseball statistics
BASIC, 27, 31, 54, 61, 128
BASIC, structured, 65
Board games, 1
Branching factor, 55

C

CBS, 5
Challenging games, 24
Checkers, 54

Chess, 22, 47, 54
Combat result table subroutine, 37
Combats, results of, 35
Complexity of games, 23
Computer artificial intelligence models, 29
Computer decision making, 47
Computer decision subroutine, Word Square, 43
Computer games, 1
Computer, interfacing with, 24
Computers as players, 41
Conflict simulations, 35
Constants, 64
Contest, balancing, 23
Contract terms, 129
Conventions, listing, 132
Conversion rights, 130
Corporations, 32

D

Deciding on game type, 26
Decision making, computer, 47
Designers, game, 128
Designing a game, 26
Detailed simulation model, 29
Difficult games, 24
Documentation, 30

Documentation, writing, 129
Dots game, 47
Dots program listing, 48

E

Easy games, 24
Ego, satisfying, 24
Evaluation function, 54
Execution speed, improving, 67

F

Features of good games, 22
Fixed-objective noninteractive simulations, 42
Flexibility, program, 63
Flowchart, Prime Time, 72
Flowchart, Takeover, 110
Flowchart, Word Square, 87
FORTH, 31, 60, 128
FORTRAN, 54

G

Game design, 26
Game type, 26
Game-design projects, 70
Games publishers, 131
Games, abstract strategy, 27
Games, adventure, 27

- Games, arcade, 26
- Games, board, 1
- Games, computer, 1
- Games, good, 22
- Games, simulation, 28
- Games, testing, 30
- Games, word, 22, 27
- General simulation model, 28
- Good play, 23

I

- Ingredients of good games, 22
- Input design, 62
- Input/output routines, 62
- Intelligence, artificial, 29
- Interfacing with the computer, 24

J

- Joysticks, 62

K

- Keyboard input, 63

L

- Language, computer, 59
- Listing conventions, 132
- Loading instructions for Prime Time, 5
- Loading instructions for Takeover, 15
- Loading instructions for Word Square, 8
- Luck, 23

M

- Machine language get-a-word, 43
- Menu subroutine, 64
- Menu, Takeover, 126
- Menus, 63
- Model, simulation, 31
- Money, making, 128
- Money, play, 1
- Moves, 55

N

- NBC, 5
- Neilsen Company, 4
- Nodes, 55

O

- Outlining program flow, 29

P

- Parameters, 64

- Pascal, 54
- Playability, 23, 31
- Players, computers as, 41
- Playing Prime Time, 5
- Playing Takeover, 16
- Playing Word Square, 8
- Plys, 55
- Poor play, 23
- Prime Time color setting routine, 65
- Prime Time game, 2, 4, 34, 64
- Prime Time program documentation, 71
- Prime Time program listing, 74
- Prime Time, playing, 5
- Producers, software, 131
- Profiteers, advice to, 128
- Program description, Prime Time, 84
- Program description, Takeover, 126
- Program description, Word Square, 108
- Program flexibility, 63
- Program flow, outlining, 29
- Program, rough, 29
- Programming language, choosing, 59
- Projects, game-design, 70
- Pseudointelligence, 41
- Publishers, 128
- Publishers, submitting games to, 129
- Punishing poor play, 23

R

- RAM, 28
- RAM, limitations imposed by, 59
- RAM, saving, 67
- Ratings wars, 4
- Real-life competitive games, 21
- Real-life simulation, 31
- Realism of games, 23
- Realism, 31
- Rewarding good play, 23
- Rights, conversion, 130
- Rights, publishing, 129
- Rough program, testing, 29
- Rough program, writing, 29
- Royalties, 130

S

- Searches, tree, 56
- Simulation model, designing, 31
- Simulation model, detailed, 29
- Simulation model, general, 28
- Simulation, statistical, 38

- Simulations, 28
- Simulations, conflict, 35
- Simulations, noninteractive, 42
- Software producers, 131
- Sort shows by power listing, 35
- Speed, improving, 67
- Statistical simulation, 38
- Steps in game design, 26
- Stock, 32
- Strategy games, 27
- Structure, program, 65
- Subroutines, Prime Time, 84
- Subroutines, Takeover, 125
- Subroutines, use of, 65
- Subroutines, Word Square, 107

T

- Takeover game, 2, 31
- Takeover main control menu, 64
- Takeover, loading instructions for, 15
- Takeover, playing, 16
- Takeover program documentation, 109
- Takeover program listing, 111
- Testing completed games, 30
- Topics of games, 28
- Tree of game positions, 55
- Type, game, 26

V

- Variables, 64
- Variables, Prime Time, 71
- Variables, Takeover, 109
- Variables, Word Square, 86
- Vocabulary program listing for Word Square, 88

W

- War games, 35
- Word games, 22, 27
- Word Square computer decision subroutine, 43
- Word Square game, 2, 5, 42
- Word Square get-a-word subroutine, 43
- Word Square loading instructions, 8
- Word Square program documentation, 86
- Word Square program listing, 97
- Word Square Vocabulary program, 88
- Writing documentation, 30

Commodore 64

Advanced Game Design

If you are intrigued with the possibilities of the programs included in *Commodore 64 Advanced Game Design* (TAB Book No. 1923), you should definitely consider having the ready-to-run disk containing the software applications. This software is guaranteed free of manufacturer's defects. (If you have any problems, return the disk within 30 days, and we'll send you a new one.) Not only will you save the time and effort of typing the programs, the disk eliminates the possibility of errors that can prevent the programs from functioning. Interested?

Available on disk for the Commodore 64 at \$19.95 for each disk plus \$1.00 each shipping and handling.

I'm interested. Send me:

_____ disk for the Commodore 64 (6423S)

_____ TAB BOOKS catalog

_____ Check/Money Order enclosed for \$19.95 plus \$1.00 shipping and handling for each disk ordered.

_____ VISA _____ MasterCard

Account No. _____ Expires _____

Name _____

Address _____

City _____ State _____ Zip _____

Signature _____

Mail To: **TAB BOOKS INC.**
P.O. Box 40
Blue Ridge Summit, PA 17214

(Pa. add 6% sales tax. Orders outside U.S. must be prepaid with international money orders in U.S. dollars.)

TAB 1923

OTHER POPULAR TAB BOOKS OF INTEREST

The Computer Era—1985 Calendar Robotics and Artificial Intelligence (No. 8031—\$6.95)

Making CP/M-80® Work for You (No. 1764—\$9.25 paper; \$16.95 hard)

Going On-Line with Your Micro (No. 1746—\$12.50 paper; \$17.95 hard)

The Master Handbook of High-Level Microcomputer Languages (No. 1733—\$15.50 paper; \$21.95 hard)

Getting the Most from Your Pocket Computer (No. 1723—\$10.25 paper; \$14.95 hard)

Using and Programming the Commodore 64, including Ready-to-Run Programs (No. 1712—\$9.25 paper; \$13.95 hard)

Computer Programs for the Kitchen (No. 1707—\$13.50 paper; \$18.95 hard)

Beginner's Guide to Microprocessors—2nd Edition (No. 1695—\$9.25 paper; \$14.95 hard)

The First Primer of Microcomputer Telecommunications (No. 1688—\$10.25 paper; \$14.95 hard)

How to Create Your Own Computer Bulletin Board (No. 1633—\$12.50 paper; \$19.95 hard)

Microcomputers for Lawyers (No. 1614—\$14.50 paper; \$19.95 hard)

Mastering the VIC-20 (No. 1612—\$10.25 paper; \$15.95 hard)

BASIC Computer Simulation (No. 1585—\$15.50 paper; \$21.95 hard)

Solving Math Problems in BASIC (No. 1564—\$15.50 paper; \$21.95 hard)

Learning Simulation Techniques on a Microcomputer Playing Blackjack and Other Monte Carlo Games (No. 1535—\$10.95 paper; \$16.95 hard)

Basic BASIC-English Dictionary for the Apple™, PET® and TRS-80™ (No. 1521—\$17.95 hard)

The Handbook of Microprocessor Interfacing (No. 1501—\$15.50 paper; \$21.95 hard)

Investment Analysis with Your Microcomputer (No. 1479—\$13.50 paper; \$19.95 hard)

The Art of Computer Programming (No. 1455—\$10.95 paper; \$16.95 hard)

25 Exciting Computer Games in BASIC for All Ages (No. 1427—\$12.95 paper; \$21.95 hard)

Programming with dBASE II® (No. 1776—\$16.50 paper; \$26.95 hard)

Lotus 1-2-3™ Simplified (No. 1748—\$10.25 paper; \$15.95 hard)

Mastering Multiplan® (No. 1743—\$11.50 paper; \$16.95 hard)

How to Document Your Software (No. 1724—\$13.50 paper; \$19.95 hard)

Scuttle the Computer Pirates: Software Protection Schemes (No. 1718—\$15.50 paper; \$21.95 hard)

Using and Programming the VIC-20®, including Ready-to-Run Programs (No. 1702—\$10.25 paper; \$15.95 hard)

MicroProgrammer's Market 1984 (No. 1700—\$13.50 paper; \$18.95 hard)

PayCalc: How to Create Customized Payroll Spreadsheets (No. 1694—\$15.50 paper; \$19.95 hard)

Commodore 64 Graphics and Sound Programming (No. 1640—\$15.50 paper; \$21.95 hard)

Does Your Small Business Need a Computer? (No. 1624—\$18.95 hard)

Computer Companion for the VIC-20® (No. 1613—\$10.25 paper)

Forecasting On Your Microcomputer (No. 1607—\$15.50 paper; \$21.95 hard)

Database Manager in MICROSOFT® BASIC (No. 1567—\$12.50 paper; \$18.95 hard)

Troubleshooting and Repairing Personal Computers (No. 1539—\$14.50 paper; \$19.95 hard)

25 Graphics Programs in MICROSOFT® BASIC (No. 1533—\$11.50 paper; \$17.95 hard)

Making Money with Your Microcomputer (No. 1506—\$8.25 paper; \$13.95 hard)

C-BIMS: Cassette-Based Information Management System for the PET® (No. 1489—\$10.95 paper; \$16.95 hard)

From BASIC to Pascal (No. 1466—\$11.50 paper; \$17.95 hard)

Computer Peripherals That You Can Build (No. 1449—\$13.95 paper; \$19.95 hard)

Machine and Assembly Language Programming (No. 1389—\$10.25 paper; \$15.95 hard)

TAB

TAB BOOKS Inc.

Blue Ridge Summit, Pa. 17214

Send for FREE TAB Catalog describing over 750 current titles in print

Commodore 64™ Advanced Game Design

George A. and Nancy E. Schwenk

**Create your own exciting
computer games . . . *plus* get three full-length, pro-quality game programs!**

Successful computer game designers George and Nancy Schwenk share their techniques and formulas for creating stimulating and challenging professional-quality games—from choosing a topic to writing, testing, and documenting your program. To provide a candid and practical view of the entire game-writing process, they have included three full-length games of their own design, written for the C-64. And, these games alone are worth far more than the price of the book! The games include:

Prime Time—A game where players compete in the cutthroat world of network TV program scheduling.

Word Square—An educational word game that pits players against the wily skills of the computer.

Takeover—A sophisticated business strategy game that challenges players to become the most powerful business tycoons of all.

The authors provide complete documentation and tips on why various programming techniques were used. Plus, they explain the advantages and disadvantages of programming games in BASIC, assembly, and FORTH languages . . . take a look at the various types of computer games . . . discuss the elements necessary for a good game . . . and give advice to help you avoid time-consuming errors or impractical projects. There's also some fascinating insight into artificial intelligence and its role in computer games.

For those who are serious about creating games for profit, the authors give realistic pointers on submitting games to publishers, tips on contracts, and even a listing of software publishers.

George and Nancy Schwenk are both professionally involved in computer software development. They have numerous professionally-published game programs to their credit.

TAB TAB BOOKS Inc.

Blue Ridge Summit, Pa. 17214

Send for FREE TAB Catalog describing over 750 current titles in print.

FPT > \$10.95

ISBN 0-8306-1923-2

PRICES HIGHER IN CANADA

1045-0685